1995

FPGA implementation of RNS structures.

Radhaselvi. Venkatesan
University of Windsor

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

FPGA Implementation Of RNS Structures

by

Radhaselvi Venkatesan

A Thesis
Submitted to the Faculty of Graduate Studies through the
Department of Electrical Engineering in Partial Fulfillment
of the Requirements for the Degree of
Master of Applied Science
at the
University of Windsor

Windsor, Ontario
December, 1994.

ISBN 0-612-01497-5

Name RADHASELVI VENKATESAN

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

ENGINEERING, ELECTRONICS AND ELECTRICAL — 0544 U·M·I

SUBJECT TERM — SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences

Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

To my parents
Gnanambigai and Venkatesan

Abstract

The objective of this thesis is to investigate the applicability of Field Programmable Gate Arrays (FPGAs) for residue arithmetic applications. FPGAs are programmable devices that can be directly configured by the end user without the use of an integrated circuit fabrication facility. They offer the designer the benefits of custom hardware, eliminating high development costs and manufacturing time. The Residue Number System (RNS) has often been proposed for custom hardware implementation of high throughput DSP algorithms. Residue arithmetic operations are easier to realize using small look-up tables, and since Xilinx FPGAs use look-up tables as configurable logic blocks, they are considered as an ideal choice for RNS based designs. Proper design techniques and effective logic optimization are the key to efficient FPGA implementation. Therefore this thesis proposes a new logic optimization algorithm for FPGA realization of residue arithmetic applications. The algorithm exploits the inherent redundancy present in representing finite rings operations using binary variables. The algorithm uses Binary Decision Diagrams to represent and manipulate logic functions. Residue arithmetic modules encoders, decoder/scalers are implemented in Xilinx FPGA with and without using the logic optimization algorithm. Comparison of results proves the effectiveness of the algorithm. The above modules are simulated and tested.

Acknowledgments

I would like to express my sincere gratitude and appreciation to my supervisor Dr. G.A. Jullien, for his invaluable guidance and constant encouragement throughout the progress of this thesis. His enthusiasm and innovatory approach was a strong source of inspiration. I would like to thank Dr.W.C.Miller for the experience gained from his courses and for his valuable suggestions. The participation of Dr. N.M. Wigley as my supervisory committee member is gratefully acknowledged.

I derive great pleasure in thanking Dimitris, for his continuous support and advice in every stage, from inception to completion of this thesis. I specially enjoyed the numerous interesting discussions I had with him.

I would also like to thank my parents, without their wholehearted support and blessings none of this would be possible. Special thanks must go to my husband Anand for his understanding, patience and enormous support throughout my Master's program. I also thank my sister Padma, her optimism and encouragement enabled me in completing this work with less stress. Thanks also goes to my parents-in-law for sharing their strength and warmth.

I am thankful to Peter Graumann (University of Calgary) and Rajeev Murgai (University of California) for their useful suggestions.

Finally, the support from my friends and VLSI lab colleagues is sincerely appreciated.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are one of the fastest growing segments of the semiconductor industry. They were first introduced in 1985, and since then they have quickly gained widespread acceptance as an excellent technology for implementing moderately large digital circuits in low production volumes.

FPGAs provide a new approach to Application Specific Integrated Circuit (ASIC) implementation that features both large scale integration and user programmability. An FPGA consists of a regular array of logic blocks that can be programmed to implement combinational and sequential logic functions, and a user-programmable routing network that provides connections between the logic blocks.

Conventional ASIC implementation technologies, such as Mask Programmed Gate Arrays and Standard Cells, require extensive manufacturing effort, taking several months from beginning to end. This results in a high cost for each unit unless large volumes are produced. On the other hand by programming an FPGA, an ASIC can be manufactured in house in a matter of hours at the cost of few hundred dollars. Thus FPGAs offer the designer the benefits of

custom hardware, eliminating high development costs and turnaround time and thereby easily evolving as a lower cost alternative for VLSI implementation of circuits. It is expected that almost one billion dollars worth of FPGAs will be sold every year by 1996, representing a significant proportion of the IC market [6].

The flexibility of FPGAs has inspired researchers to try innovative ideas for various applications. The use of FPGAs for different applications is growing steadily and this would grow faster with the improvements in the technology to allow higher capacity and greater speed performance.

1.1 The Thesis

In this thesis, a new interesting application for FPGAs is explored. Here, FPGAs are considered for residue arithmetic applications. Residue Number System (RNS) has been proven by many authors[23] to be successful in the implementation of high speed DSP applications. So far, almost all the RNS based DSP algorithms are implemented only in semi or full custom methods involving long turnaround time and high costs. It is highly desirable to produce ASICs for RNS based DSP algorithms with minimum cost and manufacturing time. Since FPGAs offer those advantages, *this thesis investigates the applicability of FPGAs for RNS based DSP applications.*

1.2 Motivation

RNS is a non-weighted integer system in which addition and multiplication can be performed without interaction between residue digits in a finite ring. It is important to note that, because of this, the problem of carry propagation is eliminated, and addition and muliplication can be conducted simultaneously and in parallel without interaction between digits. This parallelism feature invites systolic architecture[15] approach for VLSI realization of residue arithmetic operations. Targetting DSP algorithms, a bit level systolic cell was proposed by Taheri[24]. Though intensive, often computations in DSP algorithms are simple cascades of multiplication and addition, so the most important systolic cell for

DSP applications performs a simple multiply/accumulate operation. This basic systolic cell is used as a building block for developing complex DSP systems in efficient systolic architectures.

There are many attractive features in FPGAs, which encouraged the systolic residue arithmetic based implementations. The arrangement of arrays of logic blocks in FPGAs appear systolic in structure, where logic blocks can be treated as Processing Elements. Additionally some FPGAs have ample flip-flops which make them suitable for pipelined systolic architectures. Computations over finite rings are, in general, difficult to logically decompose, many references suggest the use of look-up tables to store pre-computed results, so the bit-level systolic cell is constructed using a look-up table. Xilinx FPGAs use look-up tables as configurable logic blocks, and they have abundant flip-flops to support pipelining. These favourable features make Xilinx FPGAs a currently ideal choice for implementing RNS based designs.

It is inefficient to directly retarget a full or semi-custom implementation of an RNS design to FPGA technology. In a full-custom implementation, all parts of the circuit, the logic, the routing circuitry are carefully tailored to meet a set of specific requirements. An FPGA is a pre-fabricated chip with programmable logic blocks and routing connections. There are density and performance penalties associated with user-programmable routing in FPGAs. The programmable connections, which consist of metal wire segments connected by programmable switches, occupy greater area and incur greater delay than simple metal wires. So for good FPGA implementation, the perspective is to efficiently utilize the available configurable logic blocks and the programmable routing connections between them. Minimizing the number of configurable logic blocks needed to realize a given function, is a very area-efficient solution for FPGAs. Reducing the number of CLBs is also effective in terms of reducing the routing performance penalty, as the connections between the CLBs are reduced if the number of CLBs is reduced.

Minimizing the number of CLBs requires effective logic optimization techniques. Though there are many logic optimization algorithms available in the literature, the majority of

them are concerned with optimizing random logic, while the properties of specific arithmetic used are ignored. Therefore, this thesis proposes a new logic optimization algorithm suitable for residue arithmetic applications. The algorithm exploits the inherent redundancy present in representing finite rings operations using binary variables. The redundancy arises due to the mismatch between the ring order and the power of two modulus ring in which the computation is embedded. As a result the look-up table in the bit-level systolic cell stores large numbers of don't care states. For example, in order to represent a mod-17 computation, 5 bits are required, so out of $2^5 = 32$ cases, 15 are don't cares. The objective of the algorithm is to assign appropriate values to all the don't cares, so that the overall logic function can be greatly minimized. The algorithm uses Binary Decision Diagrams to represent and manipulate logic functions. Though the algorithm is developed specifically for residue arithmetic applications, it can be used for minimizing any incompletely specified function. The algorithm is called the *Don't Cares Elimination Algorithm.*

The don't cares elimination algorithm is applied to the contents of the look-up tables present in the bit-level systolic cell and the number of CLBs required for implementation is reduced. If the number of CLBs for a single systolic cell is reduced, then the CLBs required for complex designs built using the systolic cells are greatly reduced.

1.3 Thesis Objective

To summarize, the objective of this thesis is to conduct a feasibility study on implementing RNS structures in look-up table based FPGA technology. In this thesis we also propose a new logic optimization algorithm specific to residue arithmetic applications. The implemented RNS structures are simulated and tested as a final proof of the efficacy of the new algorithm.

1.4 Thesis Organization

Chapter 2 presents background material to understand the FPGA technology. It begins by analyzing the evolution of programmable devices and then proceeds with a discussion on the main FPGA architectures, and the associated CAD flow for implementation of circuits.

Chapter 3 presents the principles of the new don't care elimination algorithm, and explains the algorithm with mod-3 addition as example.

Chapter 4 deals with the implementation strategies of RNS structures in Xilinx 4000 series FPGAs. It discusses two methods of implementation, with and without the don't care elimination algorithm, and then compares the results obtained from both of them.

Chapter 5 is concerned with the simulation and testing of the implemented RNS structures. It provides a test bench for testing the programmed FPGAs.

Finally, Chapter 6 summarizes the tasks undertaken by this thesis and shows some directions that can be taken for future research work.

Chapter2

Introduction to FPGA Technology

The aim of this chapter is to introduce the FPGA technology. We begin by analyzing the evolution of the programmable devices, and then proceed with a discussion on the programming elements, the main FPGA architectures, and the associated CAD flow for the implementation of the circuits.

2.1 Evolution of programmable devices

Programmable devices have long played a key role in the design of digital hardware. They are general-purpose chips that can be configured for a wide variety of applications. The first type of programmable device to be widely used was the *Programmable Read-Only Memory* (PROM). A PROM is a one-time programmable device that consists of a decoder and a two dimensional array of memory cells. Based on the address line inputs, the decoder selects and outputs one row of the memory on the data lines. For implementing logic functions, address lines are used as the logic circuit inputs and each data line can implement a separate logic function. PROMs are best suited for implementing memory in applications such as microcontroller based systems. The programmability feature of the ROM was enhanced and devices such as the *Erasable Programmable Read-Only Memory* (EPROM)

and the *Electrically Erasable Programmable Read-only Memory* (EEPROM), which can be erased and reprogrammed came into use.

The next generation of programmable devices are called *Programmable Logic Devices* (PLDs). The basic PLDs are *Programmable Logic Arrays* (PLAs) and *Programmable Array Logic* (PALs) devices. PLAs have an AND-plane which can give any product term of the inputs as an output and an OR plane which can generate any sum term of the product terms. Thus a PLA is ideally suitable for implementation of sum-of-products forms of Boolean expressions. The two programmable planes in the PLA are expensive to build and they introduce significant propagation delays for signals. These problems motivated industry to build PALs, which have a single programmable logic plane. In PALs, only the AND-plane is programmable and the OR-plane is fixed. So PALs are inexpensive to manufacture and they provide very high speed-performance of implemented circuits. Many finite-state-machines are built using PALs with registered (latched) outputs. PALs and PLAs are collectively referred to as *Simple Programmable Logic Devices* (SPLDs) in the literature. The logic capacity of a typical SPLD is equivalent to about 100 gates. They are best used for implementing control circuitry and 'glue' logic. SPLDs are available as One Time Programmable (OTP) devices and as reprogrammable (RP) devices.

Because of the limited logic capacity of SPLDs, denser *Complex Programmable Logic Devices* (CPLDs) were introduced by the microelectronics industry. CPLDs have a hierarchical arrangement of multiple SPLDs on a single chip, and the logic capacity of CPLDs can approach 5000 logic gates. CPLDs can also support system clock rates above 100 MHz, and are available as both OTP and RP products.

As a contrast to the structure of SPLDs and CPLDs another type of device, the *Mask Programmable Gate Array* (MPGA), was introduced. MPGAs are very high density customizable chips, and consist of rows of transistors that can be interconnected to implement a desired logic circuit. User specified connections are available both within the rows (to implement basic logic gates) and between the rows (to connect the basic gates

together). All the mask layers that define the circuitry of the chip are pre-defined by the manufacturer, except those that specify the final metal layers. These metal layers are customized during fabrication to provide the necessary interconnections. So MPGAs are not really user programmable. They are very expensive and the turnaround time is very long as they have to go through the metalization steps in a fabrication facility.

In order to use the advantages, and remove the disadvantages, of SPLDs and FPGAs, one of the semiconductor companies, Xilinx, introduced a new generation of programmable devices called *Field Programmable Gate Arrays* (FPGAs). FPGAs combine the programmability feature of SPLDs and the scalable interconnection structure of MPGAs. Presently FPGAs can support logic capacities up to 20,000 gates. Typical maximum operating speeds for FPGA implementation of circuits is in the region of 40-50 MHz. Both OTP and RP FPGA products are available.

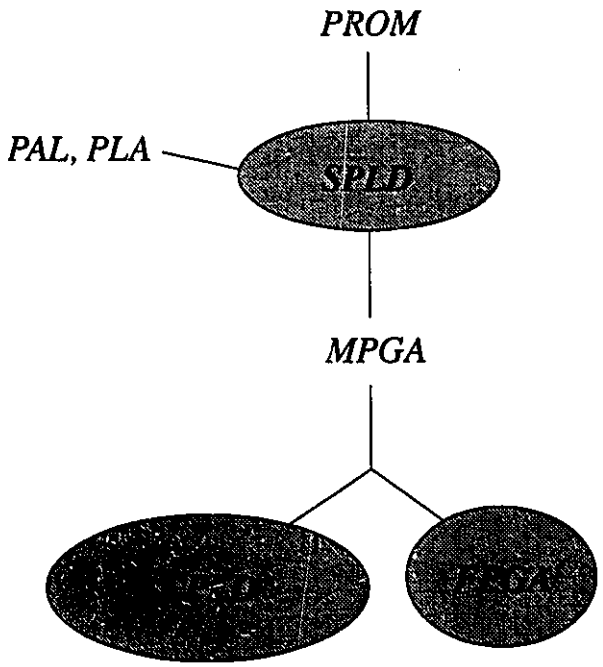


Figure 2.1 Taxonomy

The evolution of the three categories of PLDs [7] is given in Figure 2.1. Note that PROMs are not classified as PLDs, because they are mostly used only for memory applications and MPGAs are **not** PLDs, because they can be configured only during chip fabrication.

2.2 What is an FPGA?

FPGAs are programmable devices that can be directly configured by the end user without the use of an integrated circuit fabrication facility. They offer the designer the benefits of custom hardware, eliminating high development costs and manufacturing time. They were first introduced in 1985 by Xilinx. Since then, many different FPGAs have been developed by number of companies such as AT&T, Actel [1], Altera [3], Motorola, Plessey, QuickLogic, and Crosspoint Solutions. Figure 2.2 shows a conceptual diagram [6] of a typical FPGA.

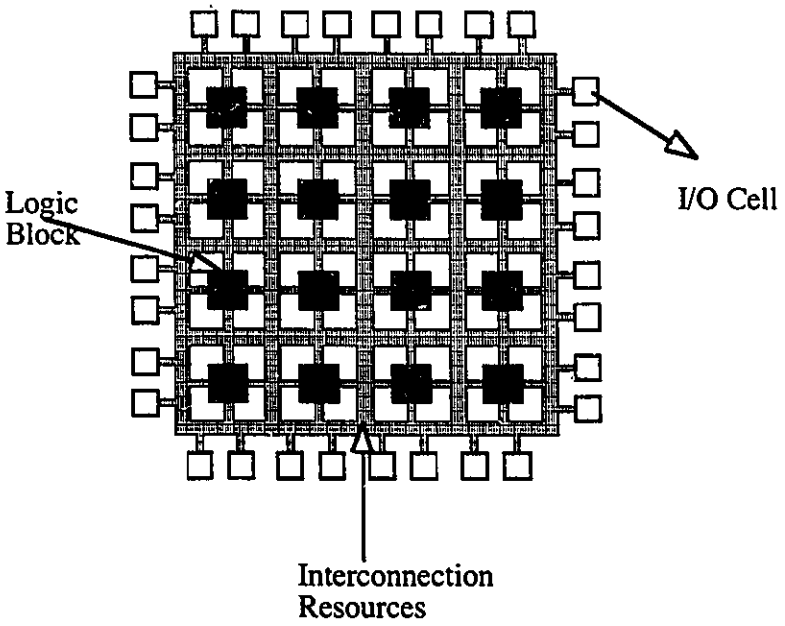


Figure 2.2 Conceptual FPGA

An FPGA generally consists of a two-dimensional array of *logic blocks* that can be connected by general interconnection resources. The interconnect comprises segments of wire, where the segments may be of various lengths. The interconnect resources include

programmable switches that serve to connect the logic blocks to one another or one wire segment to another. Logic circuits are implemented in the FPGA by partitioning the logic into individual logic blocks and then interconnecting the blocks as required via the switches.

The structure and content of a logic block are called its *architecture*. There are different kinds of logic block architecture available, and logic blocks can be built using look-up tables (Xilinx), multiplexers (Actel) or even PALs (Altera).

The structure and content of the interconnect resources in an FPGA is called its *routing architecture*. The routing architecture consists of wire segments and programmable switches. The programmable switches are constructed in several ways, including: pass-transistors controlled by static RAM cells, anti-fuses, EPROM transistors, and EEPROM transistors. There exists many different ways to design the structure of a routing architecture, some FPGAs offer simple connection between blocks, and others provide fewer, but more complex routes.

There are many advantages in using the FPGA technology and they can be used in almost all the applications that currently use MPGAs, PLDs and Small Scale Integration (SSI) logic chips. Some of the applications are listed below.

Application Specific Integrated Circuits: An FPGA can be considered as a general medium for implementation of ASICs. Some examples that have been reported [6] are: a 1 megabit FIFO controller, a DRAM controller, a graphics engine and an optical character recognition circuit.

Implementation of random logic: Random logic circuitry is usually implemented using PALs. If the speed of the circuit is not of critical concern (PALs are faster than most FPGAs), such circuits can be implemented advantageously with FPGAs. A single FPGA can implement a circuit that might require ten to twenty PALs.

Replacement of SSI Chips for Random Logic: A single FPGA can replace a number of SSI chips in existing commercial products, which results in substantial area reduction of the circuit boards that carry such chips.

Prototyping: FPGAs are very well suited for prototyping logic designs. The low cost of implementation and short time needed to physically realize a given design, provide them enormous advantages over traditional approaches for building prototype hardware.

FPGA-Based Compute Engines: FPGAs inspired a whole new class of computers, these computers consist of a board of in-circuit re-programmable FPGAs with interconnections mostly between neighbouring chips. The idea is that a software program can be "compiled" (using high level, logic level synthesis) into hardware rather than software. This hardware is then implemented by programming the board of FPGAs. This approach has two major advantages: First, there is no instruction fetching as required by traditional microprocessors, as the hardware directly embodies the instructions. This can markably increase the operating speed of the computer. Secondly, this computing medium provides high levels of parallelism, resulting in further speed increase. Using the above concept, there is evidence that, at the research level, the Digital Equipment Corporation in Paris [5] has achieved performance ranging from 25 billion operations per second up to 264 billion operations per second on applications such as RSA cryptography, the discrete cosine transform and 2-D convolution.

On-site Reconfiguration of Hardware: Some FPGAs can be reprogrammed unlimited number of times. Reprogrammability is a very attractive feature where hardware has to be changed dynamically, or where hardware has to be adapted to different user applications. A simple example is computer equipment in a remote location whose hardware has to be altered on site in order to correct a failure or perhaps a design error.

Disadvantages: The two main disadvantages of FPGAs are: their relatively low speed of operation, and relatively low logic density. The propagation delays in FPGAs is adversely affected by the inclusion of programmable switches, which have significant resistance and

capacitance, in the connections between logic blocks. A direct comparison with MPGAs indicates that a typical circuit will be slower by a factor of roughly three if implemented in an FPGA. Logic density is decreased because the programmable switches and associated programming circuitry require a great deal of chip area compared to the metal connections in an MPGA. Typical FPGAs are a factor of 8 to 12 times less dense than MPGAs manufactured in the same IC fabrication process. So at higher production volumes FPGAs become much more expensive than MPGAs.

Figure 2.3 [7] shows that, over a wide range of situations, FPGAs represent an excellent choice. Applications which require 5000 to 20,000 gates, and a volume of production up to 50,000 units, can benefit from the use of FPGAs. Even for circuits below 5000 gates, FPGAs can be considered.

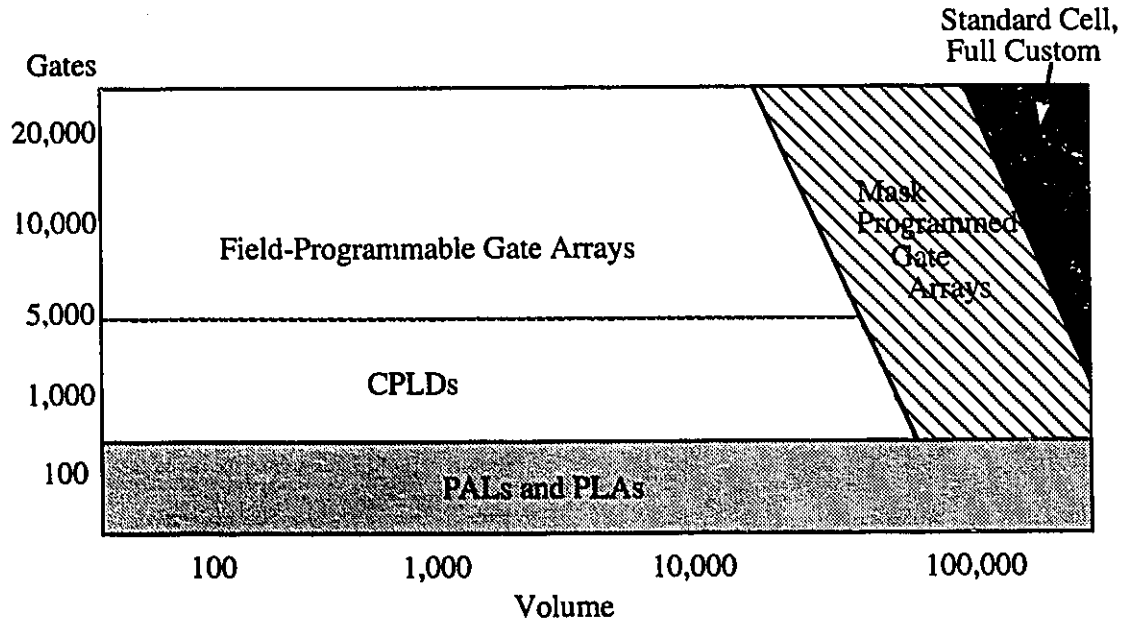


Figure 2.3 Choice of Programmable Devices with respect to gate density & volume

2.3 Programming Technologies

It is useful to gain a better understanding of how FPGAs are made field-programmable. The term *programmable switch* actually refers to the programmable elements of the FPGA chip and a typical FPGA may contain 100,000 of them. Programming elements [6] are implemented using different technologies, such as static RAM cells, anti-fuses, EPROM transistors and EEPROM transistors. Whatever the technology of implementation, the programming elements all share the property of being configurable in one of two states: ON or OFF. The desirable properties of the programming elements are

- They should consume as little chip area as possible,
- They should have a low ON resistance and a very high OFF resistance,
- The programming element should contribute low parasitic capacitance to the wiring resources to which it is attached,
- It should be possible to reliably fabricate a large number of them on a single chip.

Reprogrammability is also a desirable feature and, in terms of ease of manufacture, it might be desirable if the programming elements can be produced using standard CMOS process technology.

2.3.1 Static RAM Programming Technology

Static RAM programming technology is used in FPGAs produced by several companies: Concurrent Logic, Plessey Semiconductors and Xilinx. In these FPGAs, programmable connections are made using pass-transistors, transmission gates, or multiplexers, that are all controlled by SRAM cells. The use of a static RAM cell to control a CMOS pass transistor is illustrated in Figure 2.4.

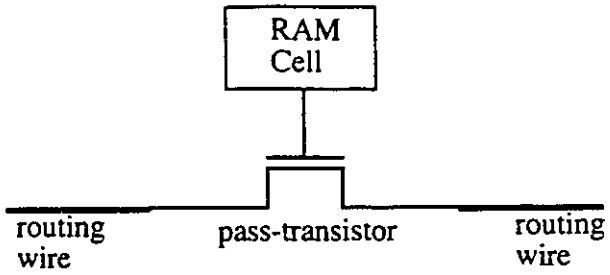


Figure 2.4 Static RAM Programming Technology

The RAM cell controls the pass-gates to be turned on or off. When off, the pass-gate presents a very high resistance between the two wires to which it is attached and the wires are thus disconnected. When the pass gate is turned on, it forms a relatively low (non-linear) resistance connection between the two wires. Since the static RAM cells are volatile, the FPGAs built using them must be configured each time power is applied to the chip. This also implies that these FPGAs can be reconfigured in-circuit. A system that includes these chips must have some sort of permanent storage mechanism for the RAM cell bits, such as a ROM or a disk, to down-load the configuration each time they are powered up. The chip area required by the static RAM approach is relatively large. This is because at least five transistors are needed for each RAM cell, along with additional transistors for the pass-gates or multiplexers. The major advantage of this technology is that it provides an FPGA that can be reprogrammed very quickly and it can be produced using a standard CMOS process technology.

2.3.2 Anti-fuse Programming Technology

Anti-fuse programming technology is used in FPGAs offered by Actel Corp., QuickLogic and Crosspoint Solutions. Although the anti-fuse used in each of these FPGAs differ in construction, their function is the same. An anti-fuse normally resides in a high-impedance state but can be "fused" into a low-impedance when programmed by a high voltage. The Actel anti-fuse, called PLICE [12], is described here. It can be described as a square structure that consists of three layers: the bottom layer is composed of positively-doped

silicon (n+ diffusion), the middle layer is a dielectric (Oxygen-Nitrogen-Oxygen insulator), and the top layer is made of poly silicon. This construction is shown in Figure 2.5.

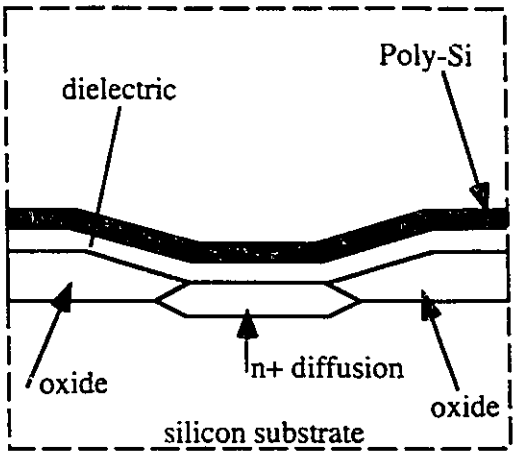


Figure 2.5 Cross section of PLICE Anti-fuse

The PLICE anti-fuse is programmed by placing a relatively high voltage (18 V) across the anti-fuse terminals and driving a current of about 5 mA through the device. This procedure generates enough heat in the dielectric to cause it to melt and form a conducive link between the Poly-Si and n+ diffusion. Special high-voltage transistors are fabricated within the FPGA to accommodate the necessary large voltages and currents.

Both the bottom and top layer of the anti-fuse are connected to metal wires, so that, when programmed, the anti-fuse forms a low resistance connection (from 300 to 500 ohms) between the two metal wires. The PLICE anti-fuse is manufactured by adding three specialized masks to a normal CMOS process.

The chip area required by an anti-fuse is very small compared to the other programming technologies. However this is somewhat offset by the large space required for the high-voltage transistors that are needed to handle the high programming voltages and currents. Anti-fuse based FPGAs are one time programmable devices. A disadvantage of anti-fuses is that their manufacture adds extra processing steps to the basic CMOS process.

2.3.3 EPROM and EEPROM Programming Technology

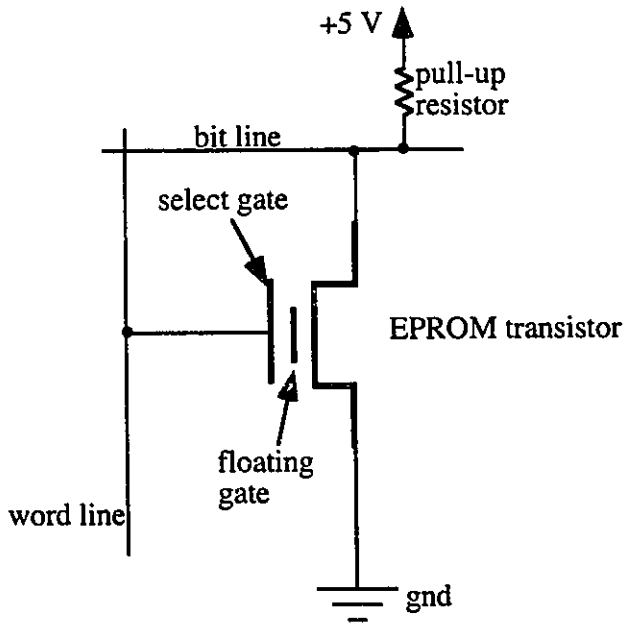


Figure 2.6 EPROM Programming Technology

EPROM programming technology is used in FPGAs manufactured by Altera Corp. [3] and Plus Logic. The technology is the same as that used in EPROM memories. Unlike a simple MOS transistor, an EPROM transistor comprises two gates, a floating gate and a select gate. The floating gate is not electrically connected to any circuitry. In its unprogrammed state, no charge exists on the floating gate and the transistor can be turned ON in the normal fashion using the select gate. However, when the transistor is programmed by causing a large current to flow between the source and drain, a charge is trapped under the floating gate. The charge has the effect of permanently turning the transistor OFF. In this way, the EPROM transistor can function as a programmable element. An EPROM transistor can be re-programmed by first removing the trapped charge from the floating gate. Exposing the gate to ultraviolet light excites the trapped electrons to the point where they can pass through the gate oxide into the substrate. So EPROM transistors are not in circuit reconfigurable. EPROM transistors in addition to serving as programmable element, can be used as "pull down" devices for logic block

inputs. This arrangement is shown in Figure 2.6. As long as the transistor is not programmed into the OFF state, the word line can cause the bit line, which is connected to a logic block input, to be pulled to logic zero.

The EEPROM approach is similar to EPROM technology except that EEPROM transistors can be re-programmed in-circuit. The disadvantage of using EEPROM transistors is that they consume about the twice the chip area of EPROM transistors and they require multiple voltage sources.

2.3.4 Summary of Programming Technologies

Table 2.1 lists some of the characteristics of the programming technologies discussed in this section.

Table 2.1. Characteristics of Programming Technologies

Programming Technology	Volatile	Re-Prog	Chip Area	R (Ohm)	C (ff)
Static RAM	yes	in circuit	large	1-2 K	10-20 ff
PLICE Anti-fuse	no	no	small anti-fuse, large program-ming transistors	300 - 500	3 - 5 ff
EPROM	no	out of circuit	small	2-4 k	10 - 20 ff
EEPROM	no	in circuit	2xEPROM	2-4 k	10 - 20 ff

The second column from the left gives an indication of whether or not the programmed element is volatile and the third column states if the element is re-programmable. Chip area is given in relative terms. The fifth column list the series resistance of the programming element in the ON state and the sixth column gives the capacitance that the element adds to each wire to which it is attached. The numbers shown in the table are for 1.2 μm CMOS. The resistance and capacitance numbers are approximate and are meant to provide a relative measure for the various elements.

2.4 Look-up table (LUT) based FPGA Architectures

This section provides a description of LUT-based architectures using the Xilinx FPGA family as an example. The general architecture of a Xilinx FPGA [6] is shown in Figure 2.7.

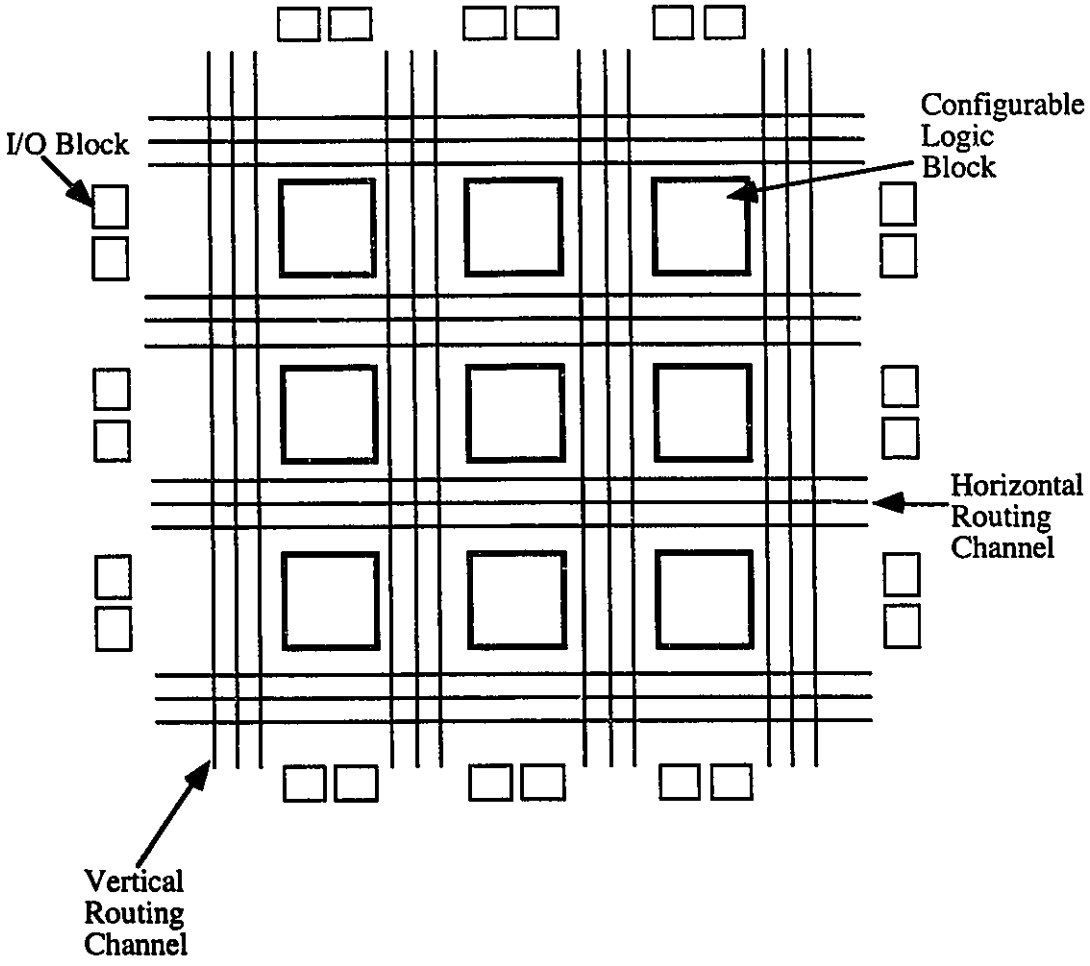


Figure 2.7 General Architecture of Xilinx FPGAs

It consists of a two-dimensional array of programmable blocks, called Configurable Logic Blocks (CLBs), with horizontal routing channels between rows of blocks and vertical routing channels between columns.

Programmable resources are controlled by static RAM cells. There are three families of Xilinx FPGAs, called the XC2000, XC3000, and XC4000 corresponding to first, second, and third generation devices. Table 2.2 gives an indication of the logic capacities of each generation by showing the number of CLBs and an equivalent gate count. The gate count measure is given in terms of "equivalent to a mask-programmable gate array of the same size." The gate count and the system clock rate shown in the table is given by the Xilinx corporation.

Table 2.2. Xilinx FPGA Logic Capacities

Family	# of CLBs	# of flip-flops	# of I/Os	Equivalent gates	System Clock Rate
XC2000	64 - 100	122 - 174	58 - 74	1,200-1,800	33 MHz
XC3000	64 - 480	256 - 1320	64 - 176	2,000-9,000	80 MHz
XC4000	64 - 576	256 - 1536	64 - 192	2,000-13,000	40 MHz

A CLB in the XC2000 family [30] consists of a 4-input look-up table with two outputs, and a D flip-flop. The look-up table can generate any function up to four variables or any two functions of three variables. Both the CLB outputs can be combinational, or one output can be registered. The XC2000 routing architecture employs three types of routing resources: direct interconnect, general purpose interconnect, and long lines. The direct interconnect provides connections from the output of a CLB to its right, top, and bottom neighbours. For connections that span more than one CLB, the general purpose interconnect provides horizontal and vertical wiring segments, with four segments per row and five segments per column. Each wiring segment spans only the length or width of one CLB. At every intersection of four CLBs, switch matrices are present, they hold a number of routing switches that can interconnect the wiring segments on its four sides. Longer wires are formed by connecting general purpose wiring segments through switch matrices. Connections that are required to reach several CLBs with low skew can use the long lines, which traverse at most one routing switch to span the entire length or width of the FPGA.

The XC3000 [30] is an enhanced version of the XC2000, featuring a more complex CLB and more routing resources. The CLB includes a look-up table that can implement any function of five variables, or any two function of four variables that use no more than five distinct inputs. The CLB has two outputs, both of which may be either combinational or registered. The XC3000 routing architecture is similar to that in the XC2000, having direct interconnect, general purpose interconnect and long lines. Each resource is enhanced: the direct interconnect can additionally reach a CLB's neighbour, the general purpose interconnect has an extra wiring segment per row, and there are more long lines.

The Xilinx XC4000 family [31] features several enhancements over its predecessors. As this family is of particular interest for this thesis work, it is explained in detail here. The XC4000 CLB is shown in Figure 2.8. The CLB has totally 13 inputs and 4 outputs. There are two 4-input function generators F,G and one 3-input function generator H. Four independent inputs F1-F4 and G1-G4, are provided to function generators F and G. Outputs from F and G and external input H1 are the inputs to function generator H. Signals from the function generators can exit the CLB on two outputs; F' or H' can be connected to the X output, and G' or H' can be connected to the Y output. With this arrangement a CLB can be used to implement any two independent functions of up to four variables, or any single function of five variables, or any function of four variables together with some functions of five variables, or it can implement even some functions up to nine variables. This implies that even for a complex function of nine variables the propagation delay is independent of the complexity and it depends only on the look-up table access time. Moreover, implementing wide functions in a single block reduces both the number of blocks required and the delay in the signal path, achieving both increased density and speed.

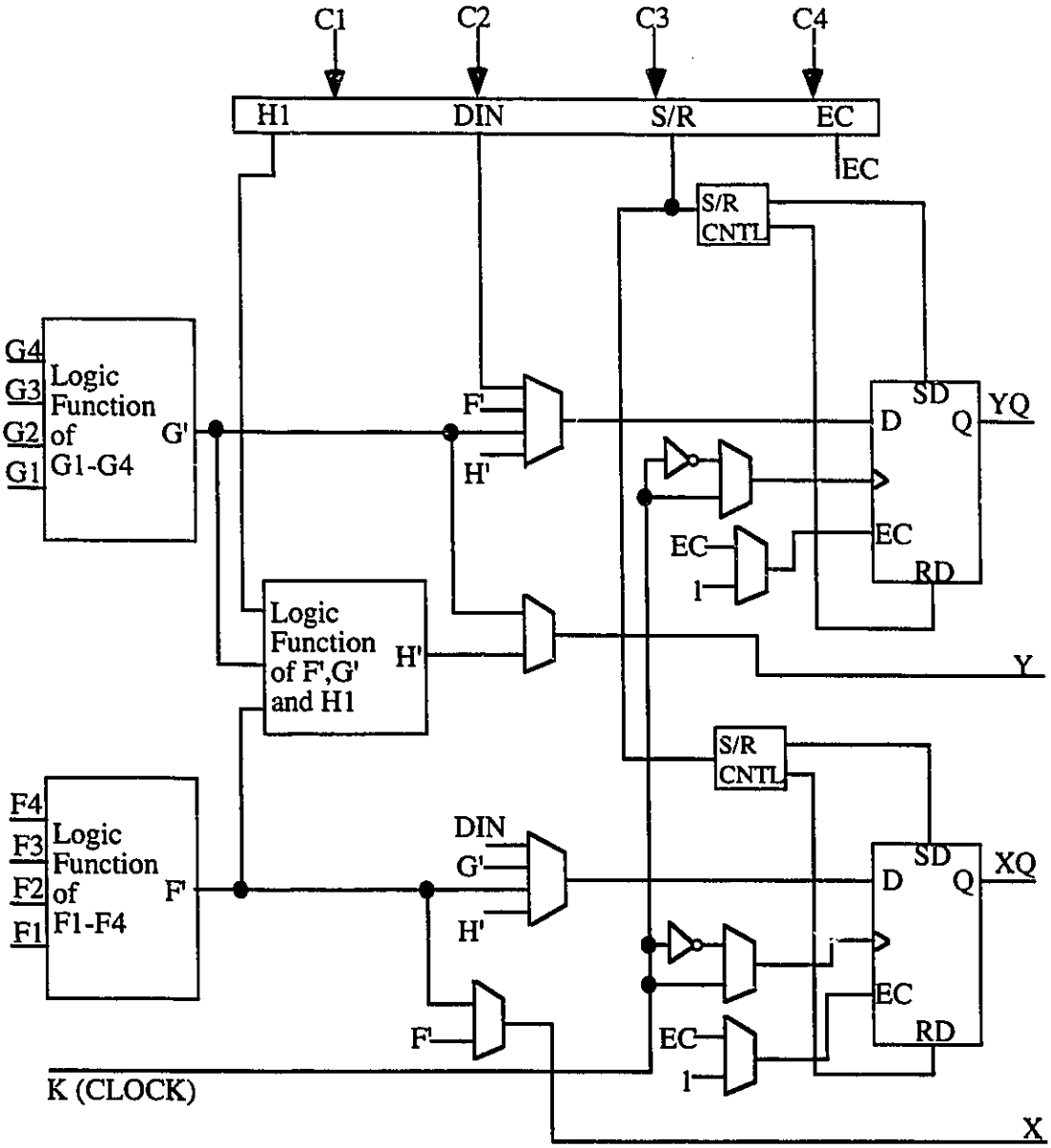


Figure 2.8 XC4000 CLB

There are two flip-flops in the CLB, both are edge-triggered D-type flip-flops with a common clock (K) and clock enable (EC) inputs. A third common input (S/R) can be programmed as either an asynchronous set or reset signal independently for each of the two registers. A separate global Set/Reset line (not shown in fig) sets or clears each register during power-up, reconfiguration, or when a dedicated Reset net is driven active.

This Reset net does not compete with other routing resources; it can be connected to any package pin of the chip, as a global reset input.

Each flip-flop can be triggered on either the rising or falling clock edge. The source of a flip-flop data input is programmable: it is driven either by the functions F', G' and H' or the Direct In (DIN) block input. According to the data sheet [31] given by Xilinx, *when a function generator drives a flip-flop in a CLB, the combinatorial propagation delay overlaps completely with the set-up time of the flip-flop.* The flip-flops drive the XQ and YQ outputs.

Multiplexers in the CLB map the four control inputs, labeled C1 through C4, into the four internal control signals (H1, DIN, S/R, and EC) in any arbitrary manner.

To improve performance, the XC4000 has two extra system oriented features: dedicated fast carry logic, and on-chip memory. Each CLB includes high speed carry logic that can be activated by configuration. The two 4-input function generators can be configured as a 2-bit adder with a built-in hidden carry that can be expanded to any length. There is evidence that this carry circuitry is so fast and efficient that it is far better than conventional speed-up methods such as carry generate/propagate. Using fast carry logic a 16-bit adder requires nine CLBs and has a combinatorial delay of 20.5ns. Fast carry [31] is particularly useful for high-speed addition operations in digital signal processing.

XC4000 CLBs include on-chip static memory resources. An optional mode for each CLB makes the memory look-up tables in the F' and G' function generators usable as either a 16 x 2 or 32 x1 bit array of Read/Write memory cells. The F1-F4 and G1-G4 inputs to the function generators act as address lines, selecting a particular memory cell in each look-up table. When the 32 x 1 configuration is selected, H1 acts as the fifth address bit. On-chip RAM is very useful for designs such as DMA counters, LIFO stacks and FIFO buffers.

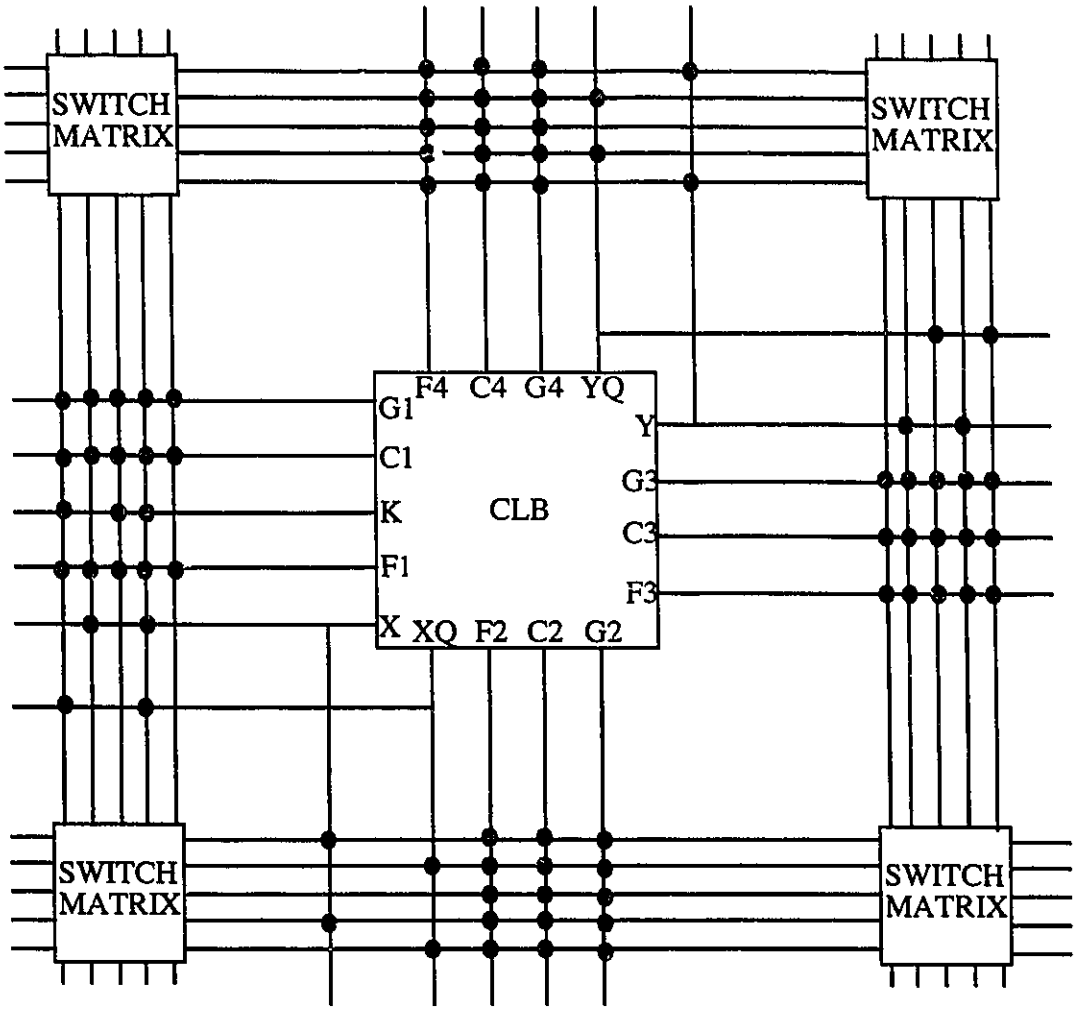


Figure 2.9 CLB connections to adjacent single-length lines

There are three types of routing resources in the XC4000 [31], they are *single length lines*, *double length lines*, and *longlines*. The *single length lines* are a grid of horizontal and vertical lines that intersect at a switch matrix between each block. Figure 2.9 illustrates the single-length interconnect lines surrounding one CLB in the array. (The number of routing channels shown in Figure 2.9 is for illustrative purposes only; the actual number of routing channels varies with array size). Each switch matrix (refer to Figure 2.10) consists of programmable n-channel pass transistors used to establish connections between single length lines. For example, a signal entering on the right side of the switch

matrix can be routed to a single -length line on the top, left, or bottom sides, or any combination thereof, if multiple branches are required.

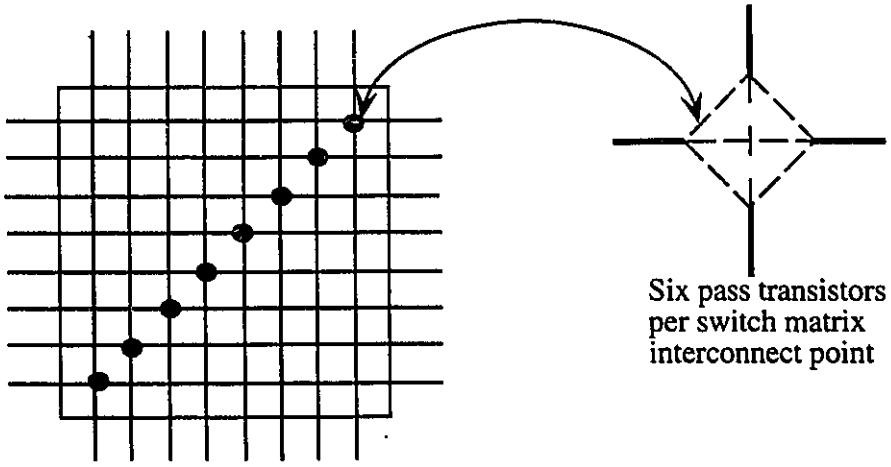


Figure 2.10 Switch Matrix

The *double length lines* consist of a grid of metal segments twice as long as the single length lines; i.e, a double length line runs past two CLBs before entering a switch matrix. Long lines form a grid of metal interconnect segments that run the entire length or width of the array. *Longlines* can be driven by global buffers, designed to distribute clocks and other high fanout control signals throughout the array with minimal skew. Communication between longlines and single length lines is controlled by programmed interconnect points at the line intersections. Double length lines do not connect to other lines.

2.5 CAD Flow

A designer who wants to make good use of FPGAs must have access to an efficient CAD system. Figure 2.11 shows the steps involved in a typical CAD system [6] for implementing a circuit in an FPGA. The system appropriate for each FPGA varies, and the one shown here is only suggestive.

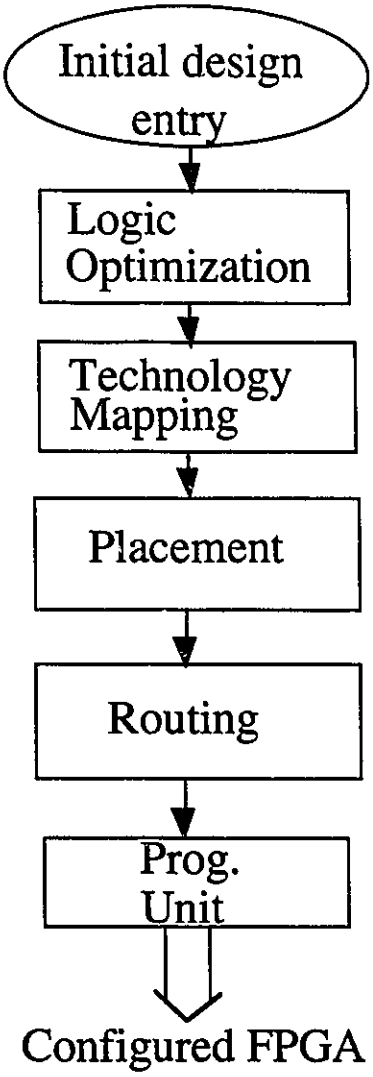


Figure 2.11 Typical CAD flow for FPGAs

The starting point for the design process is the design capture. This step may involve, drawing a schematic using a schematic capture program, entering a HDL description, or specifying Boolean equations. Whatever form the initial entry be, the circuit description is usually translated into one standard netlist form. Then logic optimization is performed on the original circuit, the goal is to modify these expressions to optimize the area or speed of the circuit or both. This optimization is technology independent and it would be appropriate for implementing a logic circuit in any medium, not just FPGAs.

The optimized circuit design must next be transformed into a circuit of FPGA logic blocks. This step is called *technology mapping* program. The mapper may attempt to minimize the total number of blocks required, which is known as area optimization. Alternatively, the objective may be to minimize the number of stages of logic blocks in time-critical paths, which is called delay optimization. To understand the technology mapping phase better, a technology mapping algorithm, realized in SIS , is explained in detail in Section 2.5.1.

Having mapped the circuit into logic blocks, it is necessary to decide where to place each block in the FPGA array, and that is done by a placement program. Typical placement algorithms attempt to minimize the total length of interconnect required for the resulting placement. The problem of placement in the FPGA environment is quite similar to that in the case of VLSI circuits implemented with standard cells.

The next step after placement is routing. The routing software assigns the FPGA's wire segments and chooses programmable switches to establish the required connections among the logic blocks. The routing software must ensure that 100 percent of the required connections are formed, otherwise the circuit cannot be realized in a single FPGA. It is also often necessary to do the routing such that propagation delays in time-critical connections are minimized. Though routing in the FPGA environment involves similar concepts as in the standard cell environment, it is complicated by the constraint that in FPGAs all of the available routing resources (wire segments and switches) are fixed in place.

Upon successful completion of the placement and routing steps, the CAD system's output is fed to a programming unit, which configures the final FPGA chip. The programming style of the FPGA varies depending on the nature of the programmable elements present in the FPGA. Actel FPGAs [1] use one time programmable anti-fuse elements. so Actel FPGAs are fused by the programming unit applying high voltage (18V). In the case of FPGAs such as the Xilinx family, where static RAM cells are used as programming elements, the configuration information can be downloaded to the FPGA serially using

standard RS-232 communication. Alternatively, a PROM or EPROM can be programmed to store the configuration information so that the Xilinx FPGA receives configuration information from them.

2.5.1 Technology Mapping in SIS

SIS [22] is an interactive tool for synthesis and optimization of combinational and sequential circuits. It is comprised of many logic optimization and technology mapping algorithms. In this section one of the SIS technology mapping algorithms [18] for look-up table based architecture is explained as an example of a family of algorithms.

From the synthesis point of view the technology mapping algorithm has these three constraints:

- A limited number of CLBs on a chip,
- Maximum number of inputs a single CLB can have,
- Limited wiring resources.

The technology mapping algorithm uses a two phase approach, the first phase is called the *decomposition* phase and the second is the *covering* phase.

Decomposition is a way of obtaining smaller nodes from a large node. Let's consider m as the number of inputs to a look-up table, then a function with m inputs or less can be realized using a single look-up table. Such a function is called m-feasible. A network is m-feasible if the function at each node of the network is m-feasible. Decomposition is used to obtain a set of m-feasible nodes for an infeasible node function. To decompose a function, classical Roth-Karp decomposition [19] is used. If decomposition is applied to a network, then the resulting network has only m-feasible nodes.

After decomposition, the algorithm now enters the covering phase. The objective of this phase is: given a feasible network, collapse nodes so that the resulting network is feasible

and the number of nodes is minimum. First, all sets of nodes which can be realized as a single block are enumerated. Each such set is called a *supernode*. The smallest subset 'S' of supernodes that follow three constraints is selected. The constraints are:

- Each node of the network should be included in at least one supernode in S.
- Each input to a supernode should either be an output of another supernode in S, or an external input to the network.
- Each output of the network should be an output of some supernode in S.

This is a binate covering formulation [20] and Mathony's algorithm [17] is used to solve this formulation.

The cost of each supernode is one, since it can be implemented by one look-up table. Now, after the covering phase, a set of supernodes is obtained with minimum total cost and each node can be mapped directly to a look-up table.

2.6 Summary

This chapter has provided an introduction to FPGA technology. Every aspect of the technology: programming elements, structural and routing architecture, CAD flow are analyzed. These issues are explained in detail for Xilinx FPGAs.

Chapter 3

Don't Care Elimination Algorithm

Binary Decision Diagrams (BDDs) are efficient and concise means of logic representation. Minimizing the size of a BDD is a crucial factor in efficient FPGA resource utilization. In this thesis, BDDs are chosen to efficiently represent and manipulate logic functions in residue arithmetic based implementations. Large numbers of don't care conditions is one prominent characteristic of residue arithmetic. Therefore, in this chapter, a new algorithm for minimizing the BDD size of incompletely specified functions is presented.

3.1 Motivation

Since computations over finite rings are, in general, difficult to logically decompose, many references suggest the use of look-up tables to store pre-computed results. Look-up tables store unminimized truth tables [4] and the truth tables have a large number of don't care outputs in the case of residue arithmetic. The excessive don't cares arise due to the mismatch between the ring order and the power of two modulus ring in which the computation is embedded. As an example, in order to represent a mod-17 computation, 5 bits are required, so out of 2^5=32 cases, 15 are don't cares. If these 15 don't cares are assigned appropriate values, then

the overall logic function can be greatly minimized. In order to exploit this property an algorithm for minimizing the BDD size of incompletely specified functions is created.

3.2 Binary Decision Diagrams

BDDs represent logic functions as directed, acyclic graphs. They were first introduced by Lee [16] and Akers [2]. Later, Randal E. Bryant proposed a new class of algorithms [8] for efficient BDD representation of logic functions. He also placed some restrictions on the ordering of the decision variables to enable efficient manipulation of BDD operations.

BDDs have several advantages over other representations such as truth tables, Karnaugh maps or canonical sum-of-products. All these approaches need exponential size representations of common functions and the processing time of the functions also grows exponentially with the size of the problem. In BDD representation a function would have, only in the worst case an exponential size graph and many of the functions encountered in typical applications have a reasonable representation. Simple operations such as satisfiability, equivalence, complementation are very time consuming in other representations. But those operations along with subtraction and addition of functions are done with ease in BDD representation. BDDs have time complexity proportional to the sizes of the graphs being operated on, so the time required for even complex operations is proportional to the product of two graph sizes.

Bryant created few rules to reduce the size of the graph, a reduced BDD for a given ordering is in its *canonical* form, i.e, every function has an unique representation.

The undesirable characteristic of BDD approach is the ordering restriction. Before processing, the ordering of the input variables to all functions has to be chosen. For some functions, the size of the graph is highly sensitive to this ordering and to find the best ordering is itself a NP-complete problem. But it is not difficult to choose an appropriate ordering by understanding the nature of the given logic function; but, there are some

functions which have a large BDD size for any possible ordering, integer multipliication is a typical example.

3.2.1 Bryant's Reduction Rules

Bryant suggested two rules for reducing the BDD size; these are the *Merge* and *Redirection* rules. Figure 3.1 shows an ordered BDD for an arbitrary logic function with a,b,c as input variables.

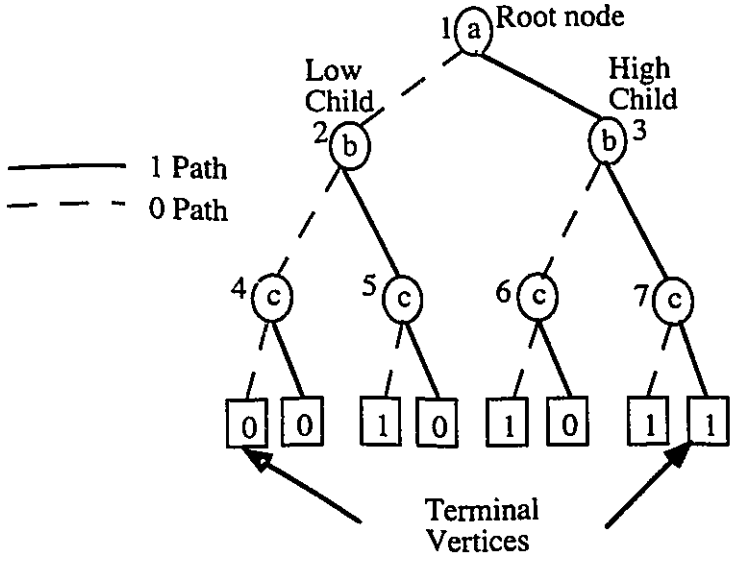


Figure 3.1 Binary Decision Diagram of an arbitrary function

The BDD has a root node with variable 'a' , the graph represents the logic function of the root node. There are two paths in the graph; the '0' path and the '1' path indicated by dotted and plain lines respectively. The low child of node 'a' is the numbered node 2 and the high child of node 'a' is node 3, i.e if the input variable a is 1, the graph takes the 1 path to node 3 and if a is 0, the graph takes the 0 path to node 2. This is true with the other nodes present in the graph. The graph ends with a set of terminal vertices.

In Figure 3.1 node 4 has the same high and low child '0' , they are considered *redundant*, so Bryant suggested that these two children can be merged together. That would be the case for node 7 too. After applying this merge rule the BDD is as shown in Figure 3.2.

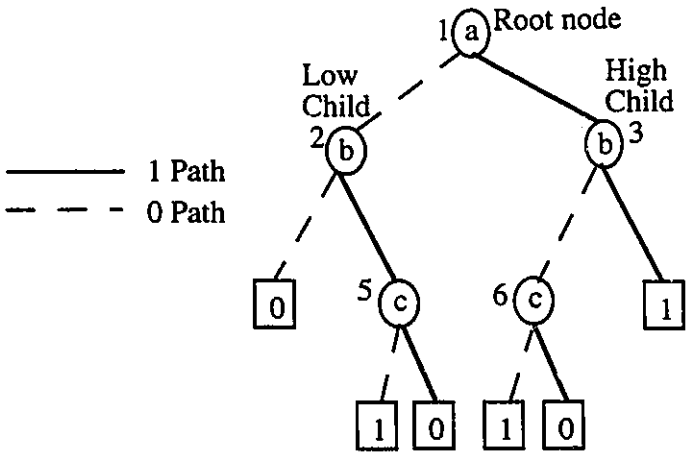


Figure 3.2 The BDD after applying the merge rule

It can be seen that Node 5 and Node 6 have similiar child nodes, but node 5 and node 6 belong to different parents, so Bryant formulated the redirection rule, the high path from node 2 will be redirected to node 6 eliminating node 5. The final BDD obtained is called a Reduced Ordered BDD (ROBDD) and it is shown in Figure 3.3.

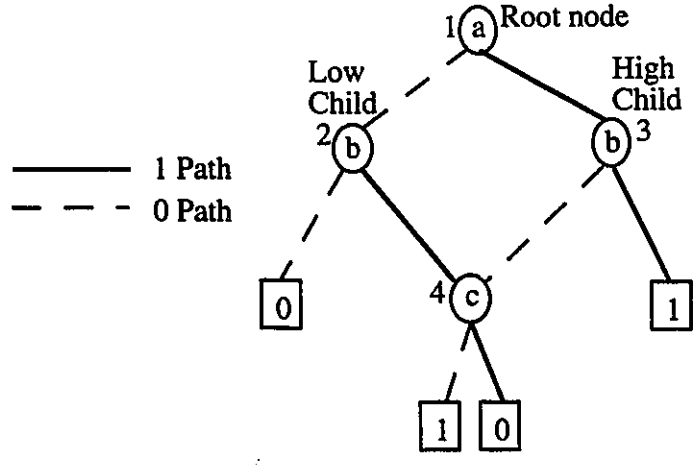


Figure 3.3 ROBDD after reduction rules

This ROBDD is an unique representation of the logic function for this particular ordering of the input variables, i.e it is in canonical norm.

3.3 Don't Care Elimination Algorithm

The algorithm adopts a divide and conquer strategy. Given a decision tree the basic idea is to create and eliminate as many *redundant* nodes as possible. This is done in a greedy manner, i.e starting from the root of the BDD the number of nodes at each level is minimized by assigning constant values to as few don't cares as necessary and propagating the constraints about the rest of them downwards. The objective is to make the two subgraphs equal as close to the root as possible.

For the purposes of illustrating the algorithm, the method is presented as a preprocessing step prior to the application of Bryant's reduction rules. The algorithm considers each incompletely specified output (don't care value) as a distinct variable and the completely specified outputs as constants. In the case of mod 3 addition (Table 3.1 on page 35) output values 0,1 and 2 are treated as three different constants and the remaining don't cares are considered as distinct variables (x_0,x_1..,x_6). The decision tree is examined at each node starting from the root, and the two children of the node, the high child and low child, are compared with each other until all the don't care variables are assigned constant values. For an n-variable function there can be a maximum of $2^n - 1$ comparisons. The operation of comparing two sub-trees and assigning appropriate values to don't cares is performed by the function *match.*

3.3.1 Match

Match [27] is a recursive function which takes two trees containing don't care variables and returns, as a result, the assignments of constants to the don't care variables, that makes the two trees as similar as possible. Matching two sub-trees fails if any of the corresponding sub-trees of them are not matchable, otherwise it succeeds by returning the possible bindings of don't cares. In case of the leaf nodes, a *match* can be defined by the four cases shown (boxes represent leaf cells), in Figure 3.4.

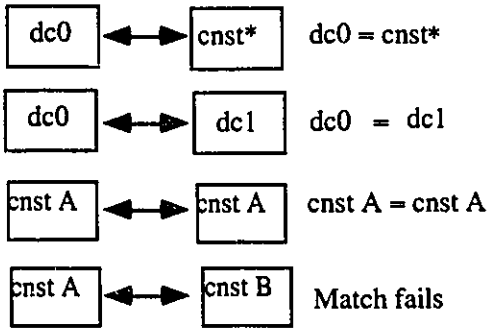


Figure 3.4 Four *Match* cases

As an example, in Figure 3.5, for a *match* between sub-trees 1 and 2, *match*(1,2), we observe that although the first three leaf cells are matchable with the result $x_2 = 0, x_0 = x_3, x_1 = 1$ the last leaf cells cannot be matched, therefore no bindings of don't cares can be made, i.e. the *match* fails.

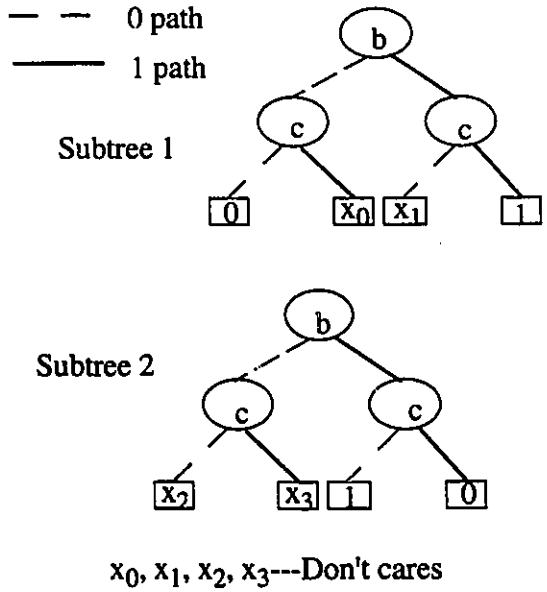


Figure 3.5 Match fail example

In the code listed for the algorithm, we represent a binary tree with decision variable and subtrees `t1` and `t2` as `Node a [t1,t2]`.

The result of function *match* is a list of associations of variables with constant values, and if this list is empty it indicates that the function *match* failed.

```
match (Node a [tsa1,tsa2]) (Node b [tsb1,tsb2]) | a==b
 = if m1==[] || m2==[] then [] -- fail if one of the matches
                                                --fails
   else m1++m2                          -- otherwise, combine results
   where
   m1 = match tsa1 tsb1
   m2 = match tsa2 tsb2

match (Const a) (Const b) | a==b     = [[]] -- success with no
                                                    bindings
match (Const a) (Var x) = [[(x,Const a)]] --   compare constants

match (Var x) (Const a) = [[(x,Const a)]] -- with don't cares

match (Var x) (Var y) = [[(y,Var x)]]   -- compare two don't cares

match a b | otherwise      = []       -- failure in all other cases
```

3.3.2 An example: Minimization of Mod3 Addition

First, the Binary Decision Tree, Figure 3.6, is constructed from the mod 3 addition truth table. (In Table 3.1 "-" indicates don't care values)

Table 3.1. Mod 3 Addition Truth Table

a1	b1	a2	b2	out
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	-
0	1	0	0	1
0	1	0	1	2
0	1	1	0	0
0	1	1	1	-
1	0	0	0	2
1	0	0	1	0

Table 3.1. Mod 3 Addition Truth Table

a1	b1	a2	b2	out
1	0	1	0	1
1	0	1	1	-
1	1	0	0	-
1	1	0	1	-
1	1	1	0	-
1	1	1	1	-

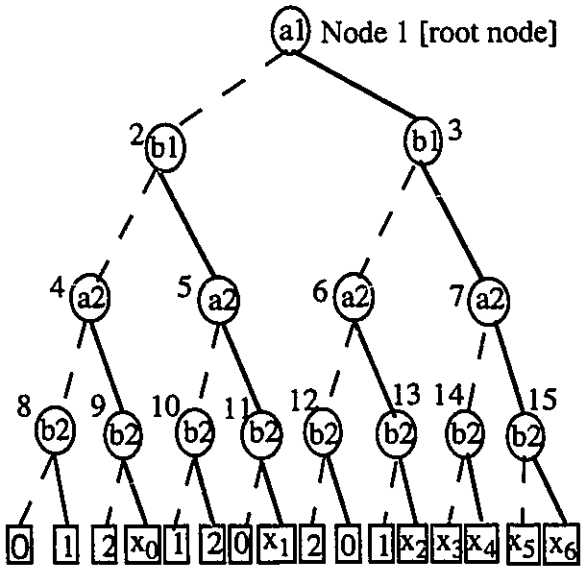


Figure 3.6 Unminimized BDD of Mod 3 Addition

Then, *match*(2,3) is performed. Since the first leaf cells are not matchable (see Figure 3.7), this fails.

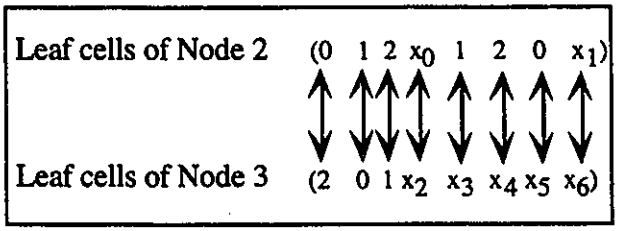


Figure 3.7 *Match*(2,3)

As the next step, *match*(4,5) and *match*(6,7) are performed. It is evident from the leaf cell values that *match*(4,5) fails and *match*(6,7) succeeds (Figure 3.8).

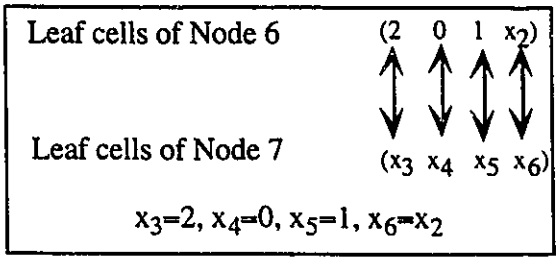


Figure 3.8 *Match*(6,7)

The BDD, after the *match*(6,7) step, is shown in Figure 3.9.

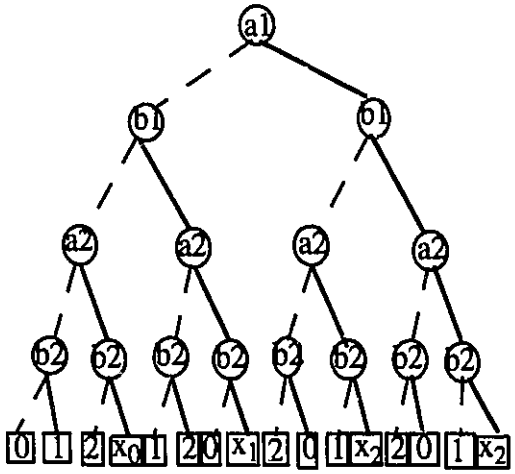


Figure 3.9 BDD of Mod 3 Addition after *Match*(6,7)

The algorithm then proceeds with matches further down the tree until all don't cares are matched. It is to be noted that under no circumstance will sub-trees, which belong to different parents, be considered for matching. The final BDD with all don't cares substituted is shown in Figure 3.10.

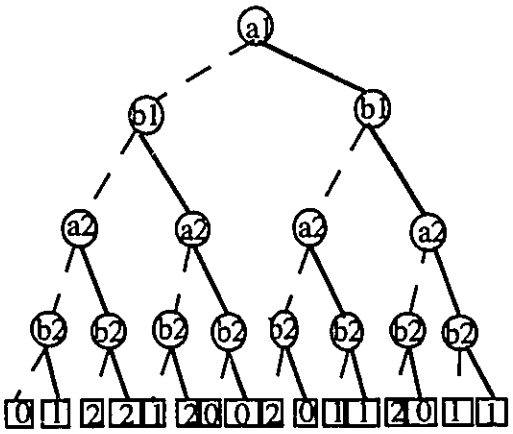


Figure 3.10 Minimized BDD of Mod 3 Addition

This BDD is reduced, with Bryant's reduction rules [8], and the resulting ROBDD is shown in Figure 3.11.

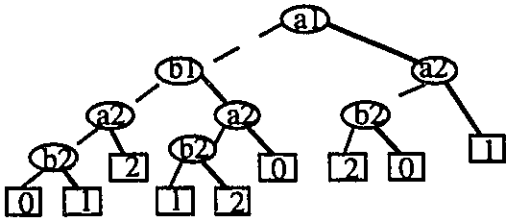


Figure 3.11 Minimized and Reduced BDD of Mod 3 Addition

To summarize, the main features of the algorithm are:

- eliminates as many redundant nodes as possible.
- never considers subtrees of different parents for *match*.
- will do a maximum of $2^n - 1$ matches for a n variable tree.
- is deterministic, for a given function gives the same result every time.
- can also be applied for multi valued functions.

In the example considered the algorithm is presented for a multi valued function, i.e the outputs are not binary but have, in this case, values of 0 (00), 1(01), 2(10). However, for better results, the algorithm can be used at the bit-level, the output of Mod-3 addition requires a 2-bit representation, and a binary decision tree for each bit can be built and the algorithm applied for both the trees.

3.3.3 Technology Independence

BDDs are, in general, technology independent representation of logic functions. Any set of logic functions represented as BDDs can be mapped to FPGAs and can be considered for full or semi custom approaches. Though the don't care elimination algorithm is developed considering an FPGA implementation, it can be very effective for implementation in other target technologies. In reference [9], H.M.Chan, in a Cascode Voltage Switch Logic (CSVL) implementation of a full adder, uses BDDs for logic representation. The basic rule[1] used by Chan treats every node as a three terminal structure where each node requires two transistors, as shown in Figure 3.12

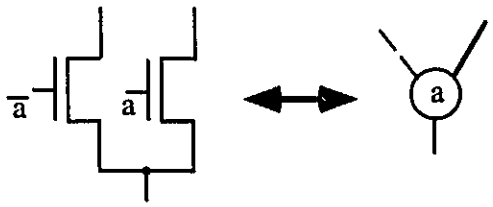


Figure 3.12 Equivalent circuit for a BDD node in CSVL

In Figure 3.12 'a' is an input variable. The discussed algorithm aims to minimize the overall size of the BDD resulting in fewer nodes and thereby producing lower transistor count. This technology independence feature is particularly useful if retargeting of design implementations between different technologies is needed.

1. The discussion here pertains to the n-block section of the CSVL.

3.4 Summary

In this chapter a new algorithm for minimizing the BDD size of incompletely specified functions is presented. The algorithm is conceptually simple, computationally inexpensive and gives very good results comparable with the only other available algorithm [10]. This algorithm has an added advantage of minimizing multi-valued functions. The algorithm discussed is a technology independent optimization procedure useful for retargeting applications.

Chapter 4

Implementation of RNS Structures

This chapter is concerned with the implementation of residue arithmetic structures on Xilinx 4000 series FPGAs. Taheri.et.al [24] [25] proposed a bit-level systolic cell which can be used as a building block for residue arithmetic based designs. The bit level systolic cell is built using ROMs and since Xilinx FPGAs use look-up tables as configurable logic blocks, they are considered as a suitable choice for implementation of RNS designs. First, the chapter explains the residue processing cell and proposes a new design methodology for implementing them in FPGAs. The new design methodology is based on the don't care elimination algorithm explained in Chapter 3. RNS modules are implemented with and without using the algorithm and the results obtained from both methodologies are assessed.

4.1 Residue Processing Cell

Though most DSP applications are computationally intensive, often those computations are simple cascades of addition and multiplications in a well defined structure. That operation of repeated multiplication and addition is performed by a generic residue processing cell, called the Inner Product Step Processor (IPSP) [14]. The $IPSP_m$ for a given moduli m computes

$$Y_{out} = Y_{in} \oplus_m [a \otimes_m X_{in}] \tag{4.1}$$

where Y is a running accumulator, a is a multiplier and X is the independent input data. All Y, a, X are B bit ring elements, $Y,a,X \in R(m)$ with $B = [\log_2 m]$. If a restriction that elements of vector a have to be fixed is imposed, then the complexity of the $IPSP_m$ cell becomes that of a simple addition. For better performance the $IPSP_m$ can be sliced at the bit level; this is called a Bit-level Inner Product Step Processor ($BIPSP_m$). A $BIPSP_m$ with a fixed multiplier can be defined by:

$$y_{i+1} = y_i \oplus_m \left[a \otimes_m x^{[i]} \otimes_m 2^i\right] \tag{4.2}$$

where i is the *spatial array index*, $y_{i+1}, y_i, a \in R(m)$ and $x^{[i]}$ is the ith bit of $X_{in} \in R(m)$.

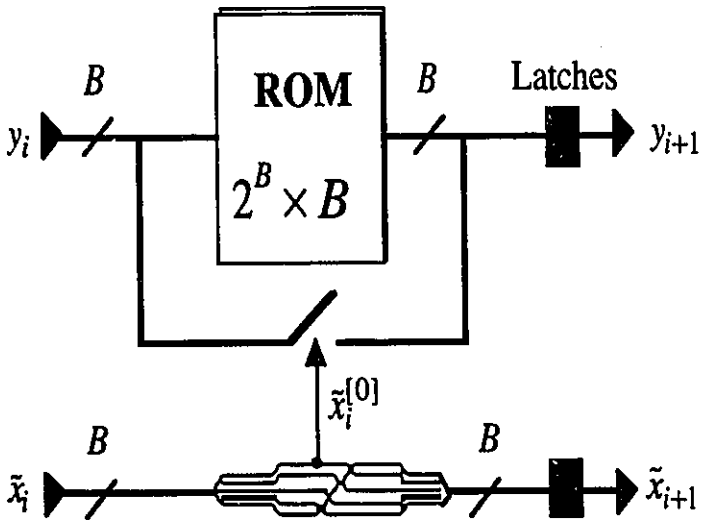


Figure 4.1 $BIPSP_m$ Cell

The implementation of the $BIPSP_m$cell is shown in Figure 4.1. Inputs to the cell are y_i and $\tilde{x}_i$, the outputs are y_{i+1} and $\tilde{x}_{i+1}$, each output line is latched. The ROM stores the operation of $y_i \oplus \left[2^i \otimes_m a\right]$. The cell computes the following relationship:

$$\text{for } x_i^{[0]} = 1, y_{i+1} = y_i \oplus_m \left[2^i \otimes_m a\right] \tag{4.3}$$

$$\text{for } x_i^{[0]} = 0, y_{i+1} = y_i \tag{4.4}$$

$x_i^{[0]}$ is known as the steering bit, since it is used to determine the direction y data should take through the cell, either through the ROM or around it. For B=5, five such $BIPSP_m$ cells are connected in cascade to get an $IPSP_m$ cell as shown in Figure 4.2

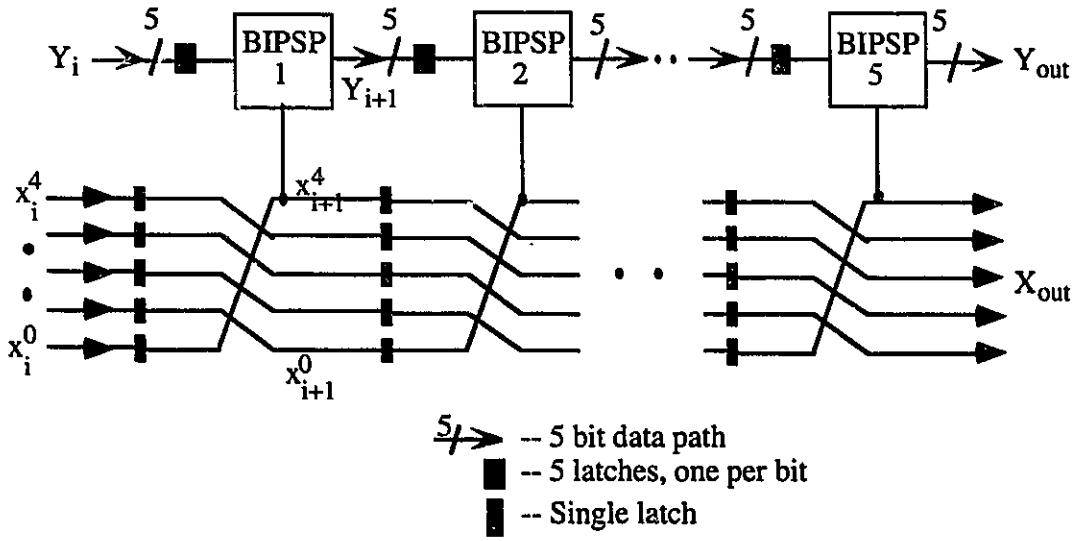


Figure 4.2 $IPSP_m$ Cell

Figure 4.2 shows that the x bits are circularly shifted by one position for each cell presenting the correct steering bits for subsequent cells when connected in a linear array.

4.1.1 IPSP$_m$ Cell in FPGA

The BIPSP$_m$ cell [24] consists of two sections, the ROM and the steering logic. Considering a 5 bit word length, each BIPSP$_m$ cell requires a 32 x 5 ROM and, 5, 2-1 multiplexers for the steering logic, as shown in Figure 4.3. In Figure 4.3, Y_i is a 5-bit word, input as address lines to the ROM, and $\hat{Y}_i$ denotes the 5-bit output of the ROM. If the steering bit x_i is 1, then the output is taken from the ROM and if the steering bit is 0, the ROM is bypassed. A 2-1 multiplexer is used for each bit of the word, with the steering bit x_i being the common multiplexing bit for all the multiplexers.

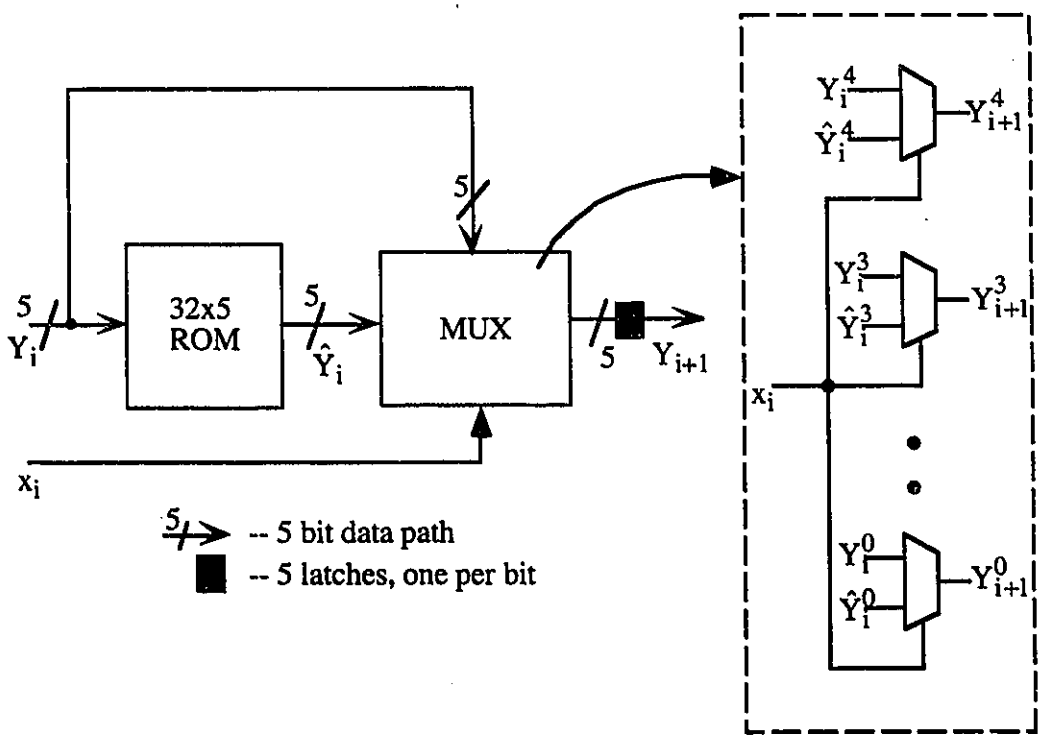


Figure 4.3 BIPSP$_m$ Cell in Xilinx FPGA

Considering a word length of B bits, B BIPSP$_m$ cells are connected linearly to form a word level IPSP$_m$cell. Figure 4.4 shows an IPSP$_m$ cell for B=5. In the FPGA implementation,

the steering bits of X are not circularly shifted. The circularly shifting scheme was proposed in reference [24] in order to obtain a regular, common cell structure for ease of routing, the trade off being the requirement for $\frac{2B}{(B+1)}$ times more latches than the minumum number. For a single $IPSP_m$ cell for B=5, the circularly shifting method requires 10 extra latches; when this $IPSP_m$ is used as a building block for designing complex operations, the number of extra latches becomes very high. In order to reduce the FPGA flip-flop count and, more importantly, the routing connections between them, the steering bits are not circularly shifted and the exact required number of latches are introduced in the X data path to maintain proper timing in the pipelined X and Y data paths.

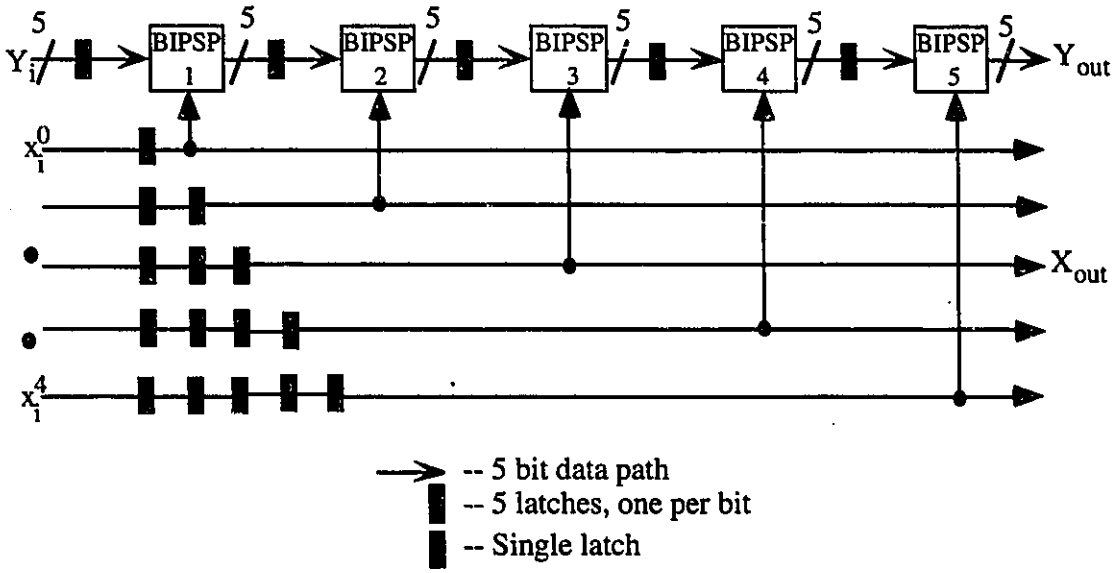


Figure 4.4 FPGA $IPSP_m$ cell

4.1.2 $IPSP_m$ Cell Implementation Methods

Two methods are presented here for implementing the $IPSP_m$ cell in a Xilinx FPGA. The first method is a *generic ROM based implementation*, which is a straight forward mapping using existing commercial software tools. In this method, a single CLB is used for implementing one output bit of the ROM. As a result, the 32x5 ROM needs 5 CLBs for implementation.

The second method is a *minimized ROM based implementation* which aims to reduce the area occupied by the ROMs by minimizing the ROMs depending on their logic function. For the ROM minimization method a separate design flow is presented, and the minimization results are merged with the rest of the $IPSP_m$ design.

4.1.3 Generic ROM based Implementation

In the XC4000 family, each CLB can be configured as either a 16x2 or a 32x1 ROM or RAM. The Xilinx software module MemGen^{TM} creates RAMs or ROMs that can be up to 32 bits wide and 256 words deep. *Memgen* prompts for the memory type, width and depth of the memory and generates memory definition files [32], *.mem* and *.xnf*. The memory type for $IPSP_m$ implementation is ROM. All the ROMs in the $IPSP_m$ design are identical in structure, they are 32 bits deep and 5 bits wide. The *.mem* file for each ROM can be edited to specify the contents of the ROM. In order to use the memory in the Cadence Composer schematic entry, a dummy symbol is created for the memory with the necessary input and output lines and the memory definition file .mem is given as a file attribute to the symbol to define the operation of the symbol.

4.1.4 Minimized ROM based Implementation

Considering the case of B=5, a 32x5 ROM in a single $BIPSP_m$cell requires 5 CLBs. The multiplexing logic for the steering operation requires 2.5 CLBs. So a single $BIPSP_m$ cell requires a total of 7.5 CLBs and a 5 bit $IPSP_m$ requires 7.5x5 = 37.5 CLBs, which implies that only 2.5 $IPSP_m$s can be implemented in a 100 CLB XC4003 chip.

For efficient implementation of the $IPSP_m$ cell, the number of CLBs occupied by a $BIPSP_m$ cell should be reduced. The number of CLBs required to implement an $IPSP_m$ cell is 37.5, and considering that there are 2 flip-flops per CLB, an $IPSP_m$ consumes the area of 75 flip-flops. To implement the pipelining section of the $IPSP_m$ cell only 40 (Y data path: 5x5 = 25, X data path 15) latches are needed. The remaining 35 flip-flops are not utilized properly. So reducing the number of CLBs will also lead to area-efficiency in flip-

flop utilization. Therefore, our method attempts to reduce the CLBs occupied by the $BIPSP_m$ cell, by reducing the area occupied by the ROM in each cell. The area of the ROM, depending on its contents, is minimized by following the following procedure:

- First the ROM contents, i.e the truth table for every output bit of the ROM, is represented as a Binary Decision Diagrams (BDDs).
- *Don't cares* in the BDDs are optimized using the *don't care* elimination algorithm (refer to Chapter 3)
- Then the traditional Reduction Rules [8] are applied to obtain a Reduced Ordered BDD (ROBDD).
- The Resulting ROBDD is mapped into Xilinx CLBs using SIS [22].

Different software programs are used for each step of the minimization procedure. The design flow is shown in Figure 4.5. The first program generates truth tables taking the required modulus and the spatial array index (i.e. the steering bit significance in the $IPSP_m$ cell) as inputs. The truth table generated is given as input to the *don't care* elimination algorithm. The algorithm is implemented in Gofer [13] - a functional programming language.

Voss [21] a package from University of British Columbia, is then used to reduce the BDD into a ROBDD. SIS [22], a CAD package from University of California, Berkeley is used for technology mapping. The SIS output is translated to the Xilinx Netlist Format (XNF) [32] using TRANS [26], developed in University of Calgary. In Appendix C, the complete design cycle from truth table to XNF generation, is illustrated with an example.

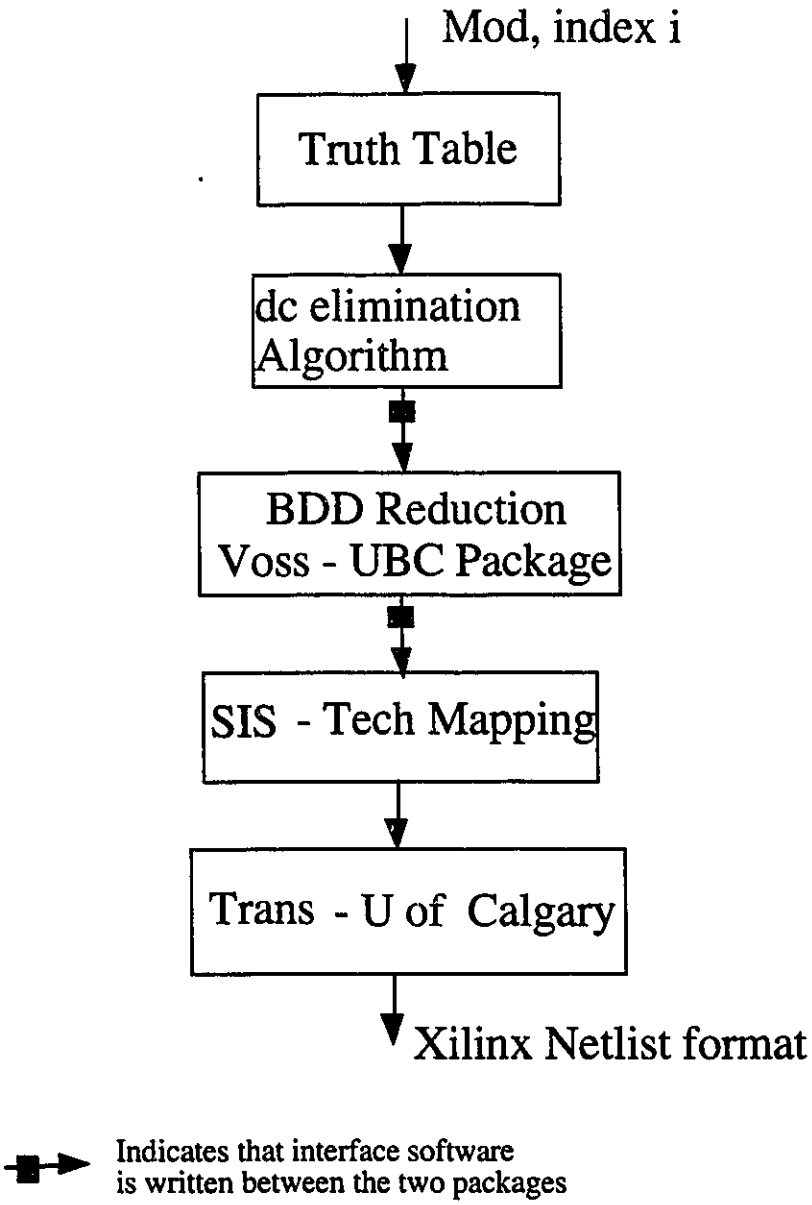


Figure 4.5 Flow chart for minimized ROM implementation method

The minimized ROM is represented in the schematic in the same manner as a generic ROM. Even here a dummy symbol is created with necessary inputs and outputs and a file attribute is attached to the symbol to define the operation of the symbol, the file attached will be the XNF file produced by the minimization procedure. This method of representation is very convenient as in typical residue arithmetic designs there may be 150

ROM output bits and using logic gates in schematics or typing Boolean expressions to represent the ROM is rather tedious.

4.2 Residue Encoder and Decoder

In this section we demonstrate our techniques using a residue modulo encoder and decoder. The encoder converts a binary number into its residue representation for a given modulus. In residue arithmetic, calculations are carried out using typically four or five moduli; in this example moduli 27, 29, 31, 32 are chosen. A scaler/decoder reduces the large dynamic range of the output resulting from residue operations using the moduli set and then combines the output to convert back to binary.

4.2.1 Encoder

The encoder maps a binary weighted number into a finite ring. An *S*-bit binary number can be mapped to a modulo *m* ring as shown in Eqn. (4.5)

$$|X|_m = \sum_{b=0}^{S-1} {}_m 2^b \otimes_m x_b \tag{4.5}$$

Taheri et.al [24] proposed an effective systolic design structure for the encoder using the $BIPSP_m$ cell as a building block. We use those results in this thesis to construct 16-bit encoders for Moduli 27,29,31,32.

The block diagram of the encoder structure is shown in Figure 4.6. In this structure, Taheri reduces the hardware required by making use of the fact that, for a 5-bit modulus conversion, the first four bits of the input binary number are already reduced by the modulus *m*.

Block B1 computes $\sum_{b=0}^{3} 2^b \otimes_m x_b \oplus_m \sum_{b=4}^{8} {}_m2^b \otimes_m x_b$, block B2 computes $\sum_{b=0}^{3} 2^b \otimes_m x_{(b+9)} \oplus_m \sum_{b=4}^{8} {}_m2^b \otimes_m x_{(b+9)}$ and block B3 combines both the results given by $\left(2^9 \otimes_m B2\right) \oplus_m B1$ to get the final encoded result. Block B1 has 5 $BIPSP_m$ cells, B2 has 3 $BIPSP_m$ cells and 2 cascades of 5 latches each and block B3 has 5 $BIPSP_m$ cells. The ROM contents of the 3 $BIPSP_m$ cells of block B2 are the same as that of the ROM contents of the first 3 $BIPSP_m$ cells of block B1.

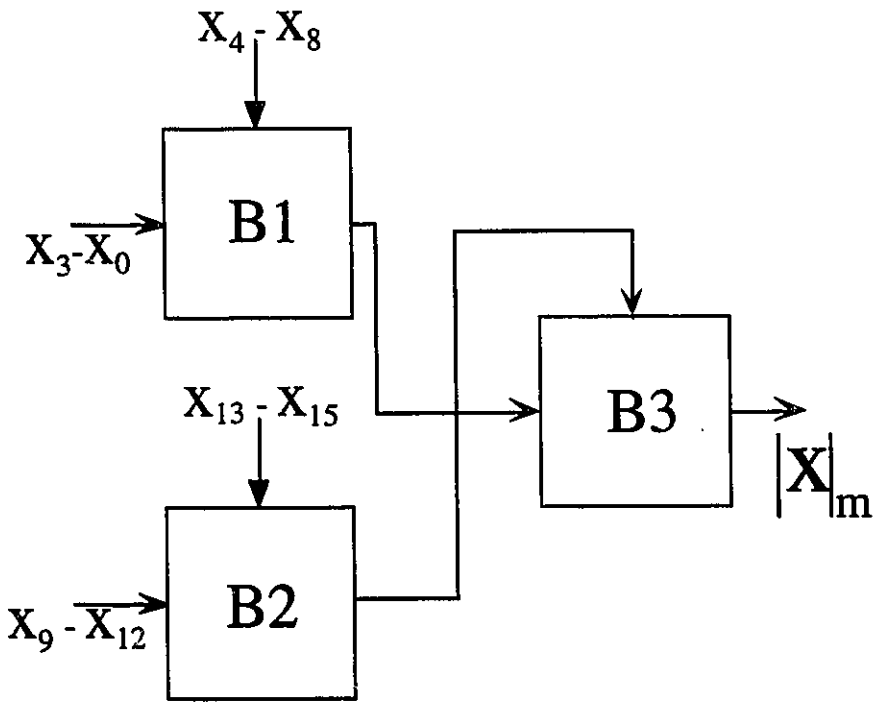


Figure 4.6 16-bit Binary to Residue Encoder

The ROM contents are given by the equation $|2^i|_m \oplus_m address$, where i is the bit significance of the steering bits. In case the binary number to be encoded is in 2' s complement form, the ROM present in the third $BIPSP_m$ cell (steered by the most

significant bit of the binary input) of block B2 stores a different truth table given by the equation $\left|-2^{i}\right|_{m} \oplus_{m} address$.

4.2.2 Decoder

Using four 5-bit moduli (27,29,31,32), a dynamic range greater than 19 bits is obtained. Scaling [14] reduces the dynamic range so that the final output is represented by a smaller number of bits. The decoder block performs the scaling and converts the residue number represented in four different moduli to its binary format. Certain restrictions in the choice of the moduli and in the order of arranging them help us to obtain an efficient implementation of the scaler/decoder block. The moduli are arranged in the order $m_1 = 29, m_2 = 27, m_3 = 31, m_4 = 32$ and scaled by 27x31. This will reduce the dynamic range to $32x29 \sim 2^{10}$.

The decoder structure is shown in Figure 4.7. Blocks B1, B2, B3, B4, B5, B6 are each made up of a single $IPSP_m$ cell (refer to Figure 4.4). The 10 output bits are obtained at the bottom of the structure with the ordering of the bits as shown in Figure 4.7. The output bits from block B5 are the 5 most significant bits of the decoder output. The least significant 5 bits of the decoder output are obtained from block B6 and hence there is a difference of 5 clock cycles between the two half of the 10 output bits. To obtain all the 10 output bits in the same clock cycle, extra latches have to be introduced for the B5 output bits. Latches are introduced in the B5 output bits path, as they also serve as steering bits for Block B6. Then, the required additional latches are added in their path in Block B6. Block B6 is shown in Figure 4.8.

The operation of the decoder blocks are defined by the following equations [14]. First

$$\vartheta_1 = m_4^{-1} \otimes_{m_1} m_3^{-1} \otimes_{m_1} m_2^{-1};\ \vartheta_3 = m_3^{-1} \otimes_{m_1} m_4^{-1};\ \vartheta_4 = m_3^{-1} \otimes_{m_4} m_2^{-1}$$

are defined. $\vartheta_1, \vartheta_3, \vartheta_4$ can be calculated in advance. Using a set of constants, as shown in equation (4.6) the input to the decoder, X1,X2,X3, and X4 can be mapped to

X_1, X_2, X_3, and X_4 employing a set of 4 pre-multipliers PM1, PM2, PM3, PM4, each consisting of a single 32x5 ROM.

$$\overline{X_1} = \left(X_1 \oplus_{m_1} \gamma \right) \otimes_{m_1} \vartheta_1$$

$$\overline{X_2} = \left(X_2 \oplus_{m_2} \gamma \right)$$

$$\overline{X_3} = \left(X_3 \oplus_{m_3} \gamma \right) \otimes_{m_3} \left| m_2^{-1} \right|_{m_3}$$

$$\overline{X_4} = \left(X_4 \oplus_{m_4} \gamma \right) \otimes_{m_4} \vartheta_4 \qquad (4.6)$$

The addition constant $\gamma = (m_1 m_2)/2$ is used to allow rounding of the estimate to the nearest integer rather than use truncation.

$$B_1 = \overline{X_3} \oplus_{m_3} \{ [-\overline{X_2}] \otimes_{m_3} m_2^{-1} \}$$

$$B_2 = \overline{X_1} \oplus_{m_1} \{ [-\overline{X_2}] \otimes_{m_1} \vartheta_1 \}$$

$$B_3 = B_2 \oplus_{m_1} \{ [-B_1] \otimes_{m_1} \vartheta_3 \} \qquad (4.7)$$

$$B_4 = \overline{X_4} \oplus_{m_4} \{ [-\overline{X_2}] \otimes_{m_4} \vartheta_4 \}$$

$$B_5 = B_4 \oplus_{m_4} \{ [-B_1] \otimes_{m_4} m_3^{-1} \}$$

$$B_6 = B_3 \oplus_{m_1} \{ [-B_5] \otimes_{m_1} m_4^{-1} \}$$

The operations of Blocks B1 to B6 are defined by eqn (4.7).

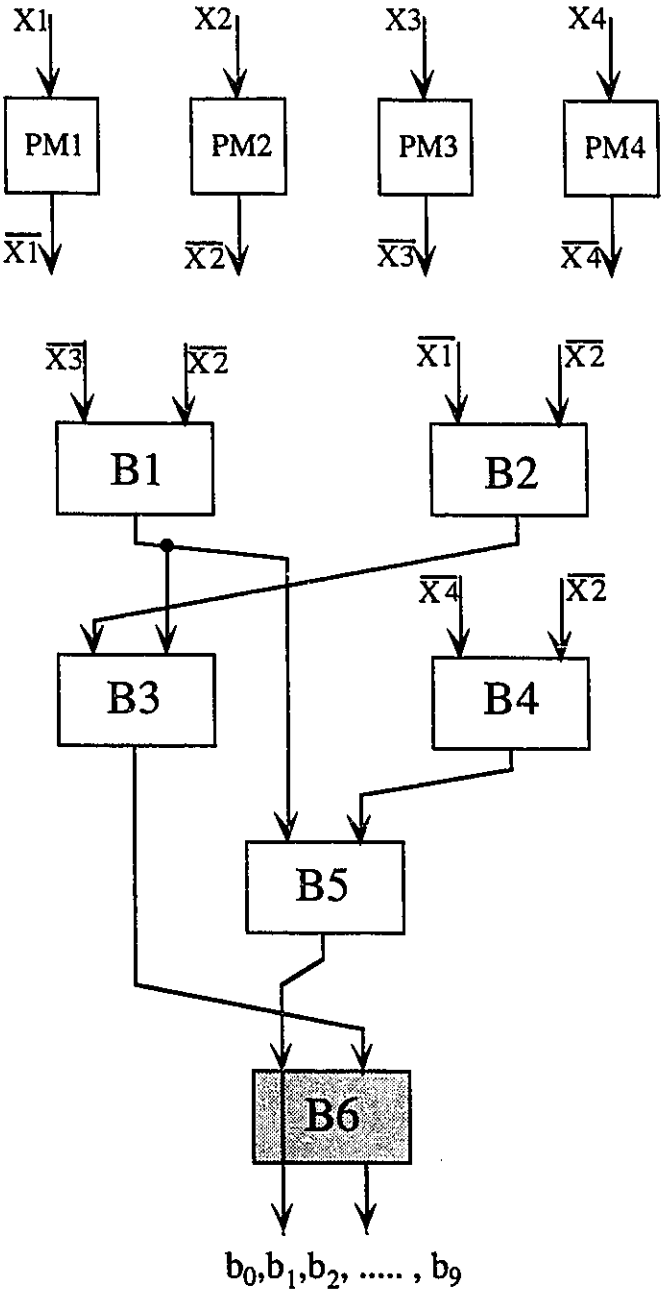


Figure 4.7 Block Diagram of the decoder

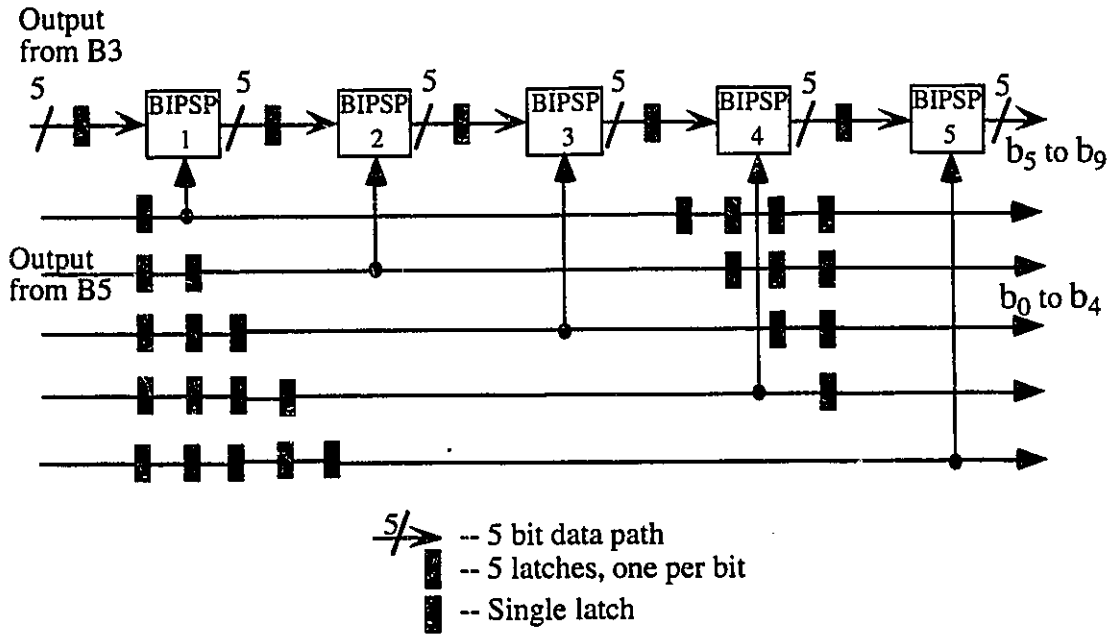


Figure 4.8 Block B6 of the decoder

4.3 Results and Discussion

16-bit encoders for moduli 27,29,31,32 and the corresponding decoder are implemented in a Xilinx 4000 series FPGA. A 9-bit encoder for Mod-17 is also considered. The designs are implemented in both generic ROM based and minimized ROM based methods. The results obtained by the two methods are presented and compared.

In the generic ROM based method, the ROMs are unminimized and they store the entire truth table. The Xilinx MemGen^{TM} program is used to generate the ROMs and the area occupied by the ROMs is predictable. Each 32x5 ROM needs 5 CLBs and the steering operation needs 2.5 CLBs so a 16-bit encoder with 13 $BIPSP_m$ cells, in generic ROM based design is expected to take 13x7.5 = 97.5 CLBs. The decoder has 6 blocks of 5 $BIPSP_m$ cells each, and 4 ROMs for the premultipliers. So (6x5x7.5) + (4x5) = 245 CLBs are necessary to implement a decoder. The results are shown in Table 4.1 . As expected the 16-bit encoders needed 98 CLBs and the decoder needed 245 CLBs.

Table 4.1. Results for Generic ROM based Implementation

Modulus	No.of CLBs	delay[a]
Encoder-27	98	35.1 ns
Encoder-29	98	34 ns
Encoder-31	98	34 ns
Encoder-32	98	35.8 ns
Decoder	245	33.5 ns

a. Longest path between any two flip-flops measured by Xdelay™ software

A 16-bit encoder with 13 $BIPSP_m$ cells, in a generic ROM based design, is expected to take 13x7.5 = 97.5 CLBs. The decoder has 6 blocks of 5 $BIPSP_m$ cells each, and 4 ROMs for the premultipliers. So (6x5x7.5) + (4x5) = 245 CLBs are necessary to implement a decoder. The results are shown in Table 4.1 . As expected the 16-bit encoders needed 98 CLBs and the decoder needed 245 CLBs.

The same designs are implemented using the minimized ROM method, in order to reduce the number of CLBs all the ROMs present in the $BIPSP_m$ cells are minimized. Results are tabulated in Table 4.2 .

As the minimization procedure involves assigning appropriate values to the don't cares, by using the don't cares elimination algorithm, the actual redundancy in the moduli and the amount of reduction in the area are checked. So the table lists the amount of reduction in CLBs along with the percentage of don't cares present in each modulus for the encoder designs. In the case of decoder the percentage of don't cares is not given, as all the four moduli are used in the design.

Table 4.2. Results for minimized ROM based Implementation

Modulus	No.of CLBs	delay[a]	Reduction in CLBs	Don't Cares
27	79	34.2 ns	19%	16%
29	86	36 ns	12%	9%
31	87	34.8 ns	11%	3%
32	57	14.7 ns	42%	0%
Decoder	181	34.9 ns	26%	-

a. Longest path delay between any two flip-flops measured by Xdelay™ software.

In the encoder for Mod-27, **19%** reduction in CLBs is achieved when compared to the result of generic ROM method. This is close to the 16% *don't cares* present in Mod-27. That holds good for moduli 29, and 31 also, where the reduction achieved is **12%** and **11%** and the amount of *don't cares* present is 9% and 3% respectively. For the decoder, a **26%** reduction in CLBs is obtained. These results prove that the minimized ROM based method is very efficient and removes the redundancy in the original logic functions for all the moduli.

To further illustrate the effectiveness of the minimized ROM based method, a 9-bit encoder for Mod-17 is also implemented in both the methods. The 9-bit encoder needs a single $IPSP_m$ cell (5 $BIPSP_m$ cells, refer Figure 4.4) for implementation. Mod-17 is specially chosen since it has the highest redundancy (47%), and the *don't care* elimination algorithm can find its best use there. In the generic ROM based method 38 CLBs are required to implement the Mod-17 9-bit encoder, but in the minimized ROM method, only 22 CLBs are required. This provides a **42%** reduction in area which is again very close to the actual redundancy present in the mod-17 operations.

The area reduction in this method is achieved by minimizing the size of the BDD representing the logic function of the ROM. In the minimization process, if the height of

the BDD is reduced from 5 variables to 4 (or less), each output bit of the ROM (32x1 locations) does not require a complete CLB. Since a CLB in the Xilinx 4000 series can be configured as a single 32x1 ROM or as two 16x1 ROMs, if the tree height is reduced to less than 5, then two output bits of the ROM can be accommodated in a single CLB. When compared with the case of the generic ROM, where only one output bit of the ROM can be obtained from a single CLB, a 50% reduction is immediately achieved.

In the case of Mod-32, when our minimization procedure is applied to the BDDs representing the ROM contents, they all reduced to single variable ROBDDs. As $2^5=32$, the binary decision trees for Mod-32 encoder, are fully reducible. Hence there is a large number of reduction in CLBs and the number of CLBs required is dictated by the flip-flops in the encoder, and not the combinational logic section of the encoder. Actually, Mod-32 needs no hardware for encoding, as the first five LSB bits of the binary input represent the encoded number.

4.3.1 Pipelining the design

The pipelining scheme is the same for both generic ROM based and minimized ROM based methods. The flip-flops present in the Xilinx 4000 series [31] are edge-triggered D-type flip-flops with common clock, clock enable and reset inputs. In the designs implemented, the reset lines of all the flip-flops are tied together. All the flip-flops can be simultaneously given an asynchronous reset at any time.

A single phase clocking scheme is used, the flip-flops are configured as rising edge triggered. In the full custom implementation of residue blocks, the clock runs against the data to avoid clock skew problems. In an FPGA implementation, we cannot ensure that the clock runs against the data as the placement and routing are done automatically, though it may be possible to control the clock routing by adopting a tedious process of manually routing the clock signal. To avoid clock skew problems, the Xilinx FPGA has special *global buffers* [31] for routing the clock signal. With an appropriate choice of those

buffers, no clock skew problems have been encountered in any of the design implementations.

4.4 Summary

This chapter has dealt with the implementation of residue building blocks, as examples of the use of FPGA technology. 16-bit encoders for moduli 27, 29, 31, 32 with a corresponding decoder, and a 9-bit encoder for Mod-17 are the designs implemented in a Xilinx 4000 series FPGA. In the implementation process, an effective logic minimization technique, which recovers the inherent redundancy in representing finite rings operations using binary variables, is created. Designs are implemented with and without the logic minimization and the results are compared. Results prove that the logic minimization procedure is very efficient, achieving good area reduction in all FPGA implementations tried.

Chapter5

Simulation and Testing

It is a common practice to simulate all the designs, in order to verify the functionality of the design and to ensure correctness in the implementation procedure. In this chapter, we present simulation results for the FPGA implementation of our RNS designs. FPGAs configured with residue designs are tested and these results are also presented in this chapter.

5.1 Simulation

Designs of the 16-bit encoders and decoder, for moduli 27, 29, 31, 32 and a Mod-17 9-bit encoder, implemented using the minimized ROM method, are simulated. The software tool used for simulation is Verilog-XL^{TM} [28]. There are two kinds of simulation, functional and timing. The following sections explain them briefly.

5.1.1 Functional Simulation

Functional simulation provides an effective method to verify the functionality of the design before the design has been placed and routed. This saves debugging time later in the design process. This is referred to as unit-delay simulation, since unit (normalized) delay times are used rather than actual delay times. Each logic gate is

modelled to have 0.1ns unit propagation delay and all nets are considered to have 0ns routing delay. The functionality of all the residue block designs considered are verified by performing functional simulation. Figure 5.1 shows the simulation result for the mod-17, 9-bit encoder. Appendix B describes in detail the procedure followed for functional simulation of the RNS designs in the FPGA implementation.

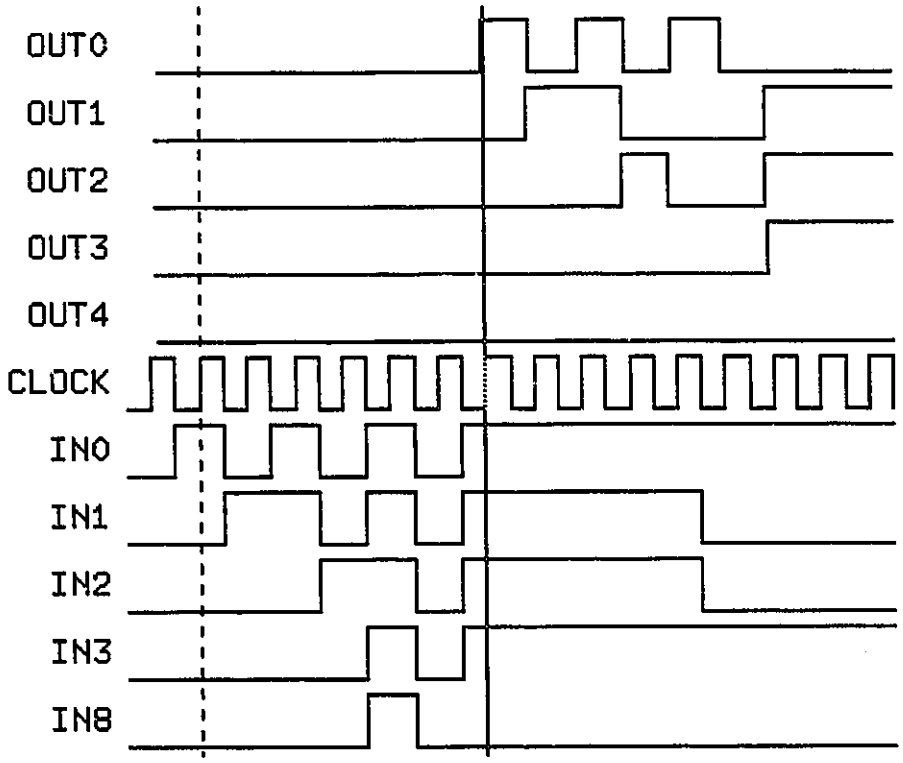


Figure 5.1 Verilog™ Simulation of the Mod-17 Encoder

5.1.2 Timing simulation

After performing functional simulation on the design, the design is mapped, placed and routed in an FPGA chip. In the mapping phase, the original design which is defined in terms of logic gates is transformed into a circuit of Configurable Logic Blocks. Then the CLBs are placed and routed in the FPGA chip. Timing simulation is performed after the design is mapped, placed and routed, so it does not use any gate models for simulation,

and it extracts the information from the placed routed file. The delay information consists of the propagation delay of the CLB blocks and the routing delays between CLBs. Since timing simulation uses worst case delay information from the routing and block delays in a placed and routed file, it provides a measure of the maximum operating speed of the design. 16-bit encoders for moduli 27, 29, 31, 32 and 9-bit mod-17 encoder are implemented in a Xilinx 4003PC84-6 chip. The decoder is implemented in a 4005PC84-6 chip. The minimized ROM method is used for all designs. A timing simulation has been performed for all the designs and Table 5.1 shows the maximum operating speed for the designs. Beyond these frequencies the circuits produce incorrect results and the set-up time of the flip-flops are violated. A detailed procedure for performing timing simulation for a Xilinx FPGA implementation is given in Appendix B.

Table 5.1. Results of Verilog-XL^{TM} Timing Simulation

Minimized ROM Designs	Maximum Operating Speed
9-bit encoder, Mod-17	30 MHz
16-bit encoder, Mod-27	26 MHz
16-bit encoder, Mod-29	26 MHz
16-bit encoder, Mod-31	27 MHz
16-bit encoder, Mod-32	48 MHz
Decoder	23 MHz

5.2 Testing

Usually an ASIC designer designs a chip and waits for a certain time period to get the final fabricated chip. FPGAs, however, are off-the-shelf products and an ASIC can be instantly manufactured in house by configuring each FPGAs with the appropriate designs. So if FPGA technology is used, the designer can test the FPGA chip immediately after designing. In this light, all of the RNS designs discussed earlier are tested. Every Xilinx FPGA chip is 100% factory tested [30] for manufacturing defects, so the testing procedure followed here is concerned only with the functionality of the implemented design.

5.2.1 XC4000 Design Demonstration Board

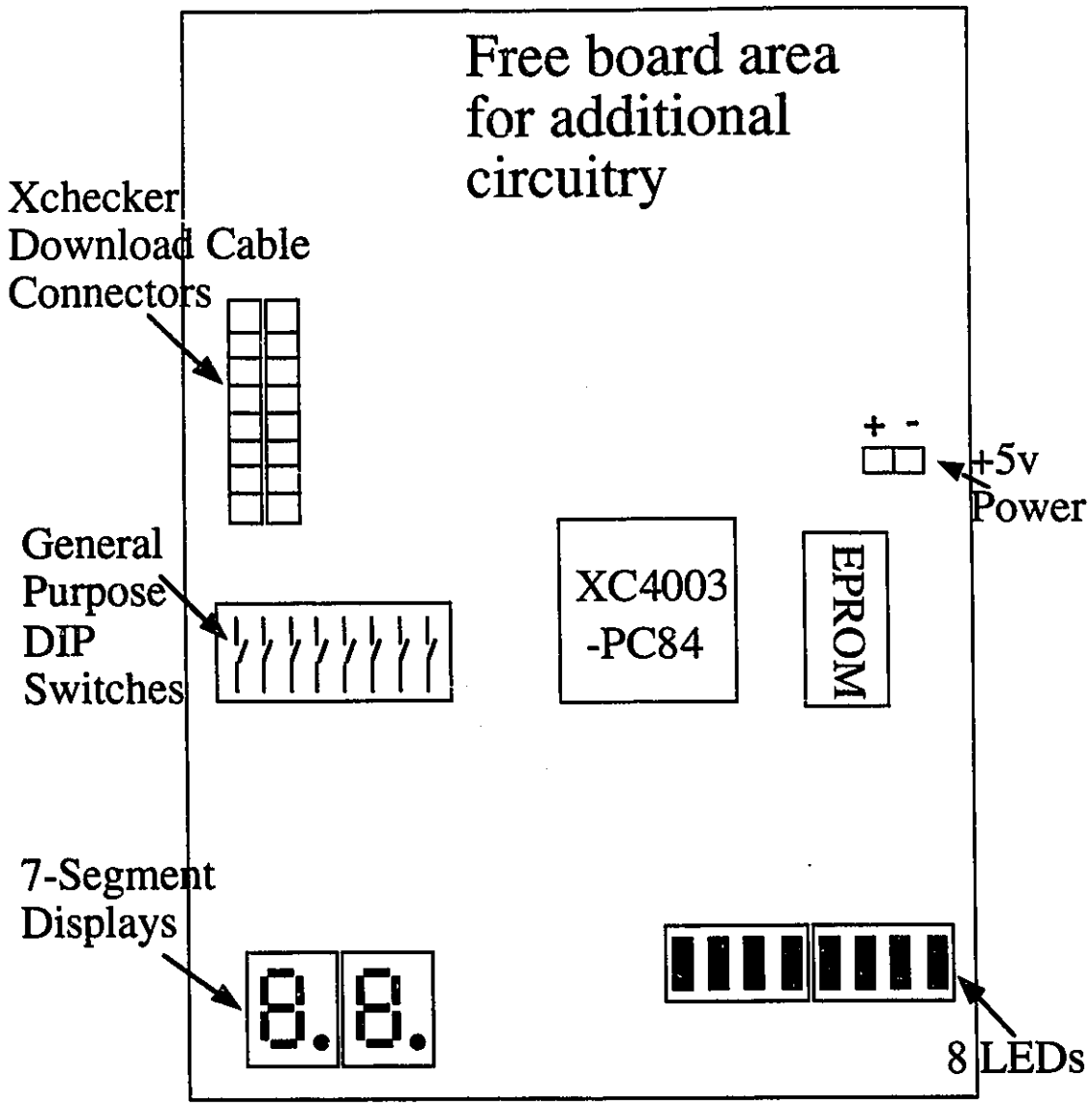


Figure 5.2 XC4000 Demonstration board

The minimized ROM based residue designs are tested using the XC4000 demo board [32], provided by the Xilinx company, the interface with the CAD system is described in Appendix A. The component layout of the board is shown in Figure 5.2. The board also contains additional components, useful for prototyping designs, including LEDs, two 7-segment displays, and two octal DIP switches. Each of these components is hard wired to

specific input and output pins. There is free space in the board area for additional circuitry. Serial EPROMs can be placed in the board if the user wishes to choose a permanent storage mechanism for the configuration data. The demo board is powered by a +5V, 200mA power supply. An *Xchecker* cable connects the demo board to the workstation, and communication is through an RS-232 serial communication link.

5.2.2 A Test Bench for the $IPSP_m$ cell

Most of the residue based designs use $IPSP_m$ cells as building blocks. For 5-bit wordlength, $IPSP_m$ cells have 5 linearly connected $BIPSP_m$ cells (see Figure 4.4 on page 46). Here the testing procedure is explained in detail for a single $IPSP_m$ cell and the same procedure can be extended to any other design built using $IPSP_m$ cells.

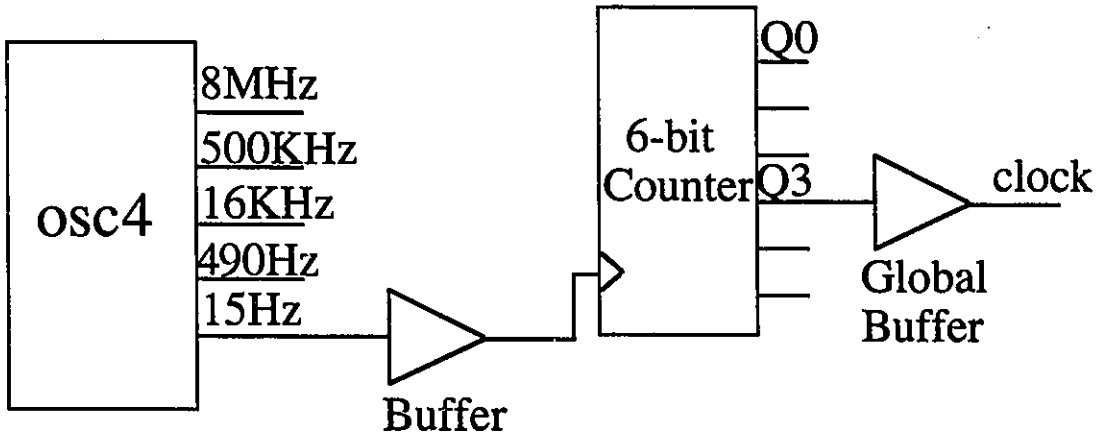


Figure 5.3 Clocking arrangement for testing

In order to test and observe the results some extra circuitry is introduced in the design of the $IPSP_m$ cell. The 5 steering inputs to the $IPSP_m$ cell are provided through the DIP switches and the outputs are observed on 7-segment displays.

An internal oscillator is available in the Xilinx 4000 series FPGAs. The oscillator can generate 4 different frequencies, they are 500 KHz, 16 KHz, 490 Hz and 15 Hz. As the outputs are manually observed in the seven segment displays, in our test, the lowest frequency of 15 Hz is chosen as the clock rate. As even 15 Hz is considerably fast for the outputs to be observed in an LED, the output from the oscillator is divided by 4 using a counter as shown in Figure 5.3. So the final clock rate at which the $IPSP_m$ cell is tested is 3.7 Hz. This testing procedure can be considered as functional testing since it does not test the chip at its operating speed.

The $BIPSP_m$ cells are designed in such a way that they are easily testable. If the steering bit input is '1' then the ROM is accessed so the ROM can be tested, if the steering bit input is '0' then the ROM is bypassed and the by-pass circuitry can be tested. The inputs to the encoder, i.e the 5 address lines to the ROM in the first $BIPSP_m$ cell, are connected to the outputs of a 5-bit counter which countinuously counts from 0 to 32, incrementing every clock cycle. The output of the $IPSP_m$ cell is visually checked in a seven segment display. The least significant 4 bits of the last $BIPSP_m$ cell are connected to a decimal to seven segment decoder and the outputs of the decoder are given to the seven segment display input lines. As in a seven segment display, only output values from 0 to 15 can be seen, the most significant bit of $BIPSP_m$ cell is observed in the decimal point of the seven segment display. Figure 5.4 gives the block diagram of the connections for testing. b0,b1,b2,b3,b4 are the steering bits.

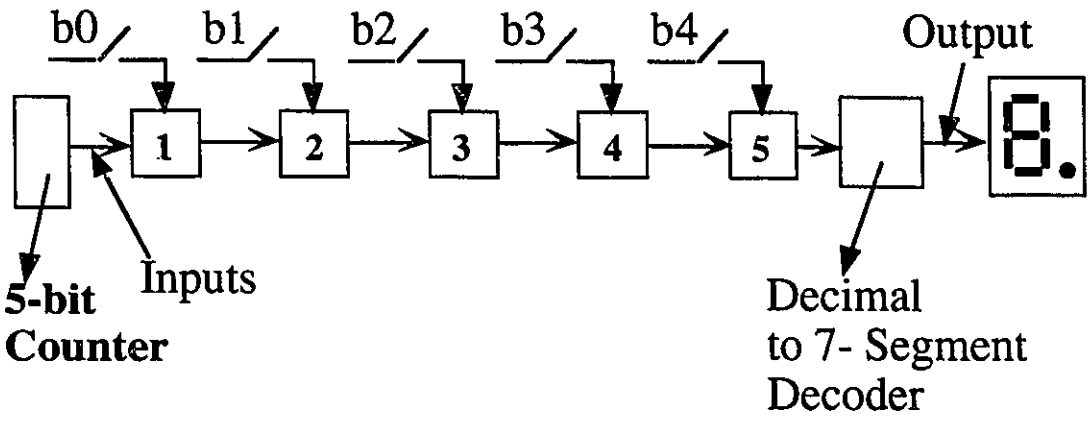


Figure 5.4 Block diagram for test bench

The steering bit can be set to '0'/'1' by turning on/off the DIP switch connected to the bit. Initially all the 5 steering bits are set to zero. The 5-bit counter supplies 32 test vectors to the input of the $IPSP_m$ cell. All the ROMs in the design are bypassed and the input to the encoder is directly passed over to the output. If the outputs are correct then the bypass circuitry in all the ROMs is functioning correctly. Next the steering bit b0, steering the first $BIPSP_m$ cell is set to 1, with all other steering bits being 0. This tests the ROM of the first $BIPSP_m$ cell. We then individually set bits b1 through b4 to logic 1. Finally all the steering bits are set to "1" and the output is checked. For each pattern of the resulting 7 DIP switch settings, 32 different inputs are given, resulting in 7x32 = 224 test vectors. A single $IPSP_m$ cell has 10 inputs, so an exhaustive test requires 1024 test vectors, but with our test procedure 224 test vectors are enough to test the chip thoroughly.

Taheri in his work [24] on testing the full custom implementation of $IPSP_m$ cell, proves that 2x32 = 64 test vectors suffice to test the entire chip thoroughly. The two cases he considered were: all the steering bits at "0" and all the steering bits at "1". But in his work, Taheri uses generic ROMs and the *don't cares* in the ROM contents were filled with zeros. Even in the minimized ROM based FPGA implementation 64 test vectors would test the chip thoroughly, but since each don't care is assigned an appropriate value by the *don't care* elimination algorithm, all the locations of every ROM is observed and that requires

224 test vectors. However when the total functionality of the chip is of primary interest, 64 test vectors are sufficient for testing a single $IPSP_m$ cell.

This testing procedure is adopted for testing the 9-bit Mod-17 encoder, the 16-bit encoders for moduli 27, 29, 31, 32 and the decoder. One of the desirable features of the Xilinx FPGA is its reconfigurability. The same chip can be programmed any number of times with different designs. So first, the chip is configured as a 16-bit Mod-27 encoder, and tested. The same 4003 chip in the demo board is configured with the other designs subsequently and all of them are tested. The XC4003 has 100 CLBs and the entire decoder needs more than 100 CLBs, so decoder is divided into 3 sections for testing.

5.3 Summary

In this chapter we have presented simulation results for the residue block designs. We have also discussed testing procedures for the FPGA implementation. The residue designs of a 9-bit Mod-17 encoder, together with encoders for moduli 27,29,31,32 and the decoder are simulated and tested and found to function correctly.

Chapter6

Conclusions

6.1 Conclusions

The main objective of this work was to explore the implementation of RNS structures in look-up table based FPGAs. That objective was completely fullfilled and in that process the following tasks were accomplished:

1. An efficient algorithm for minimizing the BDD size of incompletely specified functions was developed.
2. An effective logic minimization procedure for FPGA implementation was established.
3. Using the logic minimization procedure the following designs were implemented: Mod-17 9-bit encoder; 16-bit encoders for moduli 27, 29, 31, 32; and a corresponding decoder. All the designs were simulated and tested.
4. To compare the implementation results obtained using the minimization procedure, the same designs were implemented using an unminimized generic ROM based method. The comparison demonstrates that the minimization procedure is very effective and produces significantly better results than the unminimized method.

6.2 Desirable Features in FPGAs

Based on the experience gained in executing the work presented in this thesis, some architectural improvements for FPGAs, which will make them more efficient for ASIC implementation, are suggested.

- Heterogeneous logic blocks

Almost all the present FPGAs have homogeneous logic blocks, better performance can be expected with two or more different kinds of logic blocks. In an FPGA with an array of identical 4-input look-up tables, a 4-input function and a smaller 2-input function would occupy the same area. Instead, if the FPGA is made up of 4-input and 2-input look-up tables, the most appropriate logic block can be chosen during technology mapping phase, and this would result in more area-efficient implementations.

Having a mixture of smaller and medium sized look-up tables is particularly suitable for systolic array architectures. As pipelining is an essential feature for systolic architectures and the logic functions between two flip-flops is seldom wide, smaller sized look-up tables can be effectively used.

- Hard-wired connections

The RC delay associated with the programmable switches, brings down the performance of the FPGA. To improve the speed, some programmable connections can be replaced with metal wires and the associated CAD system can be modified to identify places to employ hard wires. With this arrangement there may be some increase in the speed of operation, and also improvements to density.

- Application specific architectures

Tuned FPGA architectures for different applications is a very desirable feature. A logic block which performs multiply, accumulate operation would be very suitable for DSP applications.

- System oriented features

System oriented features such as fast carry arithmetic circuitry, on-chip RAM, can significantly increase the performance of the FPGA.

- Partial reconfigurability

Reconfigurable FPGAs are the ideal choice for prototyping applications; even if a slight change is made to the original design, all the internal memory cells of the FPGA have to be configured again. Partial reconfigurability would be useful for such applications. A practical example is a fixed coefficient FIR filter, where an FPGA can be partially reconfigured if the coefficient set has to be changed.

- Versatile CAD system

A designer may wish to use two or more types of FPGAs for different applications. Under such circumstances a single CAD system, which can be used for a varietys of FPGAs, can save the designer time and money.

6.3 Directions for Future Work

Finally a few suggestions for future work are listed here.

1. It is worth while to explore and establish a design methodology for testing Xilinx FPGAs in the HP-VXI environment. This would help to test the FPGA at its maximum operating speed. The configuration information should be stored in a EPROM or EEPROM, so that while testing, the power to the chip can be switched off, without downloading the information to the chip every time.
2. It would be a good exercise, to investigate the implementation of RNS structures in more resource rich AT&T ORCA FPGAs. AT&T FPGAs were recently introduced with a single logic block consisting of four, 4 input look-up tables and, four flip-flops [7].

The maximum gate density available is up to 26,000 gates. Using the ORCA series, it may be possible to economically implement hardware expensive residue arithmetic based FIR filters.

3. In this thesis, logic optimization was only concerned with the combinational section of the circuits. The optimization techniques can be extended to the sequential portion. This can be done following the approach in SIS [22], where signal dependencies across register boundaries are also exploited. It would also be interesting to try Jean Vuillemin's methodology for sequential synthesis. He constructs *Synchronous Decision Diagrams* (SDDs) [29], to extend the BDD reduction techniques to sequential circuits, and he targets implementation in bit-serial architecture [11].

4. The *don't care* elimination algorithm developed is technology independent, so in future full custom implementation of RNS structures, the algorithm can be applied for minimizing the transistor count.

REFERENCES

[1] Actel, "FPGA Data Book and Design Guide." 1994.

[2] S. B. Akers, "Binary Decision Diagrams," IEEE Trans. on Computers, Vol. C-27, pp. 509-516, June 1978.

[3] Altera Data Book, 1993.

[4] M.A. Bayoumi, G.A. Jullien, W.C. Miller, "A Look-up Table VLSI Design Methodology for RNS Structures Used in DSP Applications," IEEE Trans. on Circuits and Systems, CAS-34 No. 6, June 1987.

[5] P. Bertin, D. Roncin, J. Vuillemin, "Programmable Active Memories: Performance Measurements," in FPGA '92, ACM/SIGDA First International Workshop on Field-Programmable Gate Arrays, Berkely, CA, February 1992, pp. 57-59.

[6] Stephen D. Brown, Robert J. Francis, Jonathan Rose, Zvonko G. Vranesic. "Field-Programmabl Gate Arrays." 1992.

[7] Stephen D. Brown, "Tutorial Overview on Technology, Architecture, and CAD Tools for Programmable Logic Devices.", 1994.

[8] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, Vol C-35, No. 8, pp. 677-691, August 1986.

[9] H.M. Chan, "Dynamic Logic Synthesis With Application to Self-Timed Pipelines," M.A.Sc. Thesis, University of Windsor, 1992.

[10] S-C. Chang, D-T. Cheng and M. Marek-Sadowska, "Minimizing ROBDD Size of Incompletely Specified Multiple Output Functions," EDAC-94.

[11] Peter Denyer and David Renshaw, "VLSI Signal Processing: A Bit-Serial Approach." 1985.

[12] E. Hamdy, J. McCollum, S. Chen, S. Chiang, S. Eltoukhy, J. Chang, T. Speers and A. Mohsen, "Dielectric Based Antifuse for Logic and Memory ICs," International Electron Devices Meeting Technical Digest, 1988, pp. 786-789.

[13] Mark P. Jones, "Gofer - Functional Programming Environment." 1991.

[14] G. A. Jullien, "Number Theoretic Techniques in Digital Signal Processing." Advances in Electronics and Electron Physics. Academic Press 1991.

[15] H.T. Kung, "Why Systolic Architectures," IEEE Computer Magazine, Vol. 15, No.1, pp. 37-46 Jan. 1982.

[16] C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs," Bell. Syst. Tech. Journal., Vol. 38, pp. 985-999, July 1959.

[17] H.J. Mathony, "Universal Logic Design Algorithm and its Applications to the Synthesis of Two-level Switching Circuits," IEE Proceedings, 136 pt. E(3), May 1989.

[18] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, A. Sangiovanni Vincentelli, "Logic Synthesis for Programmable Gate Arrays," In Proceedings of the Design Automation Conference, pp. 620-625, June 1990.

[19] P.J. Roth and R.M. Karp, "Minimization over Boolean Graphs," IBM Journal of Research and Development, Vol. 6, No. 2, April 1962.

[20] R. Rudell, "Logic Synthesis for VLSI Design," Memorandum No. UCB/ERL M89/49, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, April 1989.

[21] Carl-Johan H. Seger, "Voss - A Formal Hardware Verification System User's Guide," Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.

[22] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, Alberto Sangiovanni Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Electronics Research Laboratory, Memorandum No. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[23] M.A. Soderstrand, W.K. Jenkins, G.A. Jullien, F.J. Taylor. "Residue Number System Arithmetic: Modern Applications in Digital Signal Processing." 1986.

[24] M.Taheri, "VLSI Fault Tolerant Systolic Architectures." Ph.D. Dissertation, University of Windsor 1988.

[25] M. Taheri, G.A. Jullien, W.C. Miller, "High-Speed Signal Processing Using Systolic Arrays Over Finite Rings," IEEE Journal on Selected Areas in Communications. Vol. 6, No. 3, pp. 504-512, 1988.

[26] Trans User's Guide, The University of Calgary, Department of Electrical Engineering, 1993.

[27] R. Venkatesan, D. Phoukas, G.A. Jullien, "A New Algorithm for Minimizing the BDD Size of Incompletely Specified Functions," In Proceedings Canadian Workshop on Field-Programmable Devices, pp.2.6.1-2.6.5, June 1994.

[28] Verilog-XL^{TM} Reference Manual, Online Documentation.

[29] Jean E. Vuillemin, "On circuits and Numbers," Internal Report, DEC, PRL, France 1993.

[30] Xilinx, "The Programmable Logic Data Book." 1993.

[31] Xilinx, "The XC4000 Data Book." 1992.

[32] Xilinx, "The XACT Development System Reference Guide." 1993.

Appendix A

Designing with Xilinx FPGAs

This appendix discusses the CAD system available in the VLSI Lab, for design implementation on Xilinx FPGAs. The system is not a single vendor solution, but is composed of two different environments: Cadence-OPUS version 4.2 and XACT Development System™ version 4.36. Cadence is supplied by Cadence Design Systems, Inc. and XACT is provided by Xilinx Inc. Even for implementing a simple circuit, the designer has to navigate through long manuals from different vendors; therefore, to ease the learning process for other users of the Xilinx system, this appendix combines necessary information from different sources to serve as a simplified single document for the future user's reference.

A.1 Xilinx Design Flow

This section gives an outline of the Xilinx design flow for XC4000 implementation. The design entry method adopted is schematic drawing, Cadence Composer™ is used. The schematic entry procedure is no different from the common method used for any other technology. The necessary library elements are provided by Xilinx. Xilinx has three different families of FPGAs and there are three sets of libraries, one for each family. The design flow for XC4000 implementation is shown in Figure A.1. Once the schematic is entered, the software Make Netlist is executed which translates the schematic information into the format specified by Xilinx, Xilinx Netlist Format (XNF). The output file has the suffix *xnf*. Then the program PPR, (Partition[1], Place and Route) performs technology mapping and places and routes the design, giving the output in a *.lca* (Logic Configurable

1. Xilinx refers to technology mapping as partitioning

Array) file. The MakeBits software with the *.lca* file as input, produces a *.bit* file, the bit file contains the configuration data for the internal static RAM cells of the FPGA chip.

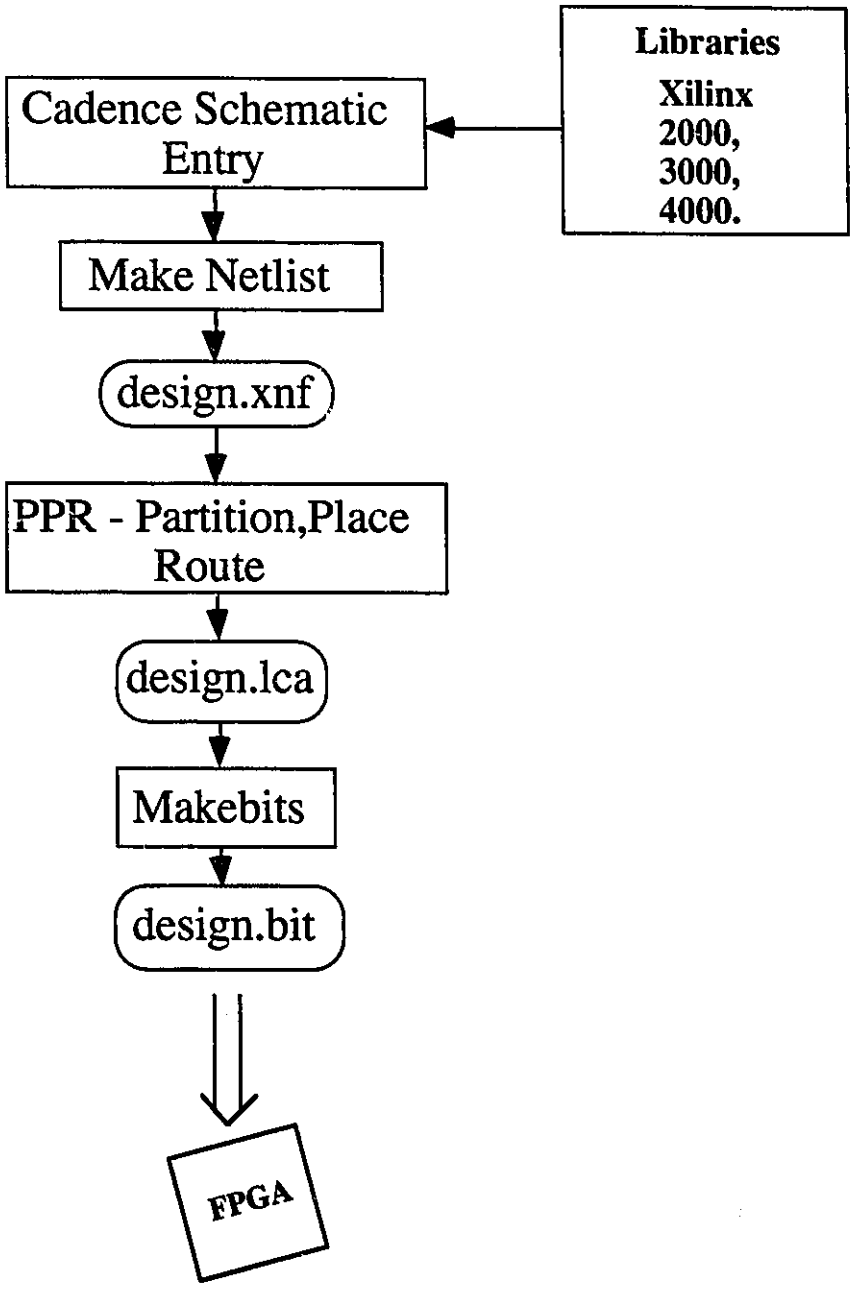


Figure A.1 Xilinx Design Flow

To use the Xilinx design flow, the user should log into the system in the Xilinx-OPUS design environment. The necessary environment variables ($XACT directory) and the path settings are done in the *allusers.login, allusers.cshrc file in the /local/login directory*, no customizing is needed in the .cshrc file present in the user's home directory.

To start the Cadence environment the user should type two commands:

```
xhost +
```

This command checks whether there are any access restrictions for the user, if none, the following message is printed,

```
all hosts being allowed (access control disabled)
```

If there is any access restriction, the user has to contact the system administrator. The next command is to start Cadence with Xilinx technology option, it is given by,

```
startCds -t xilinx
```

Now Cadence is opened and the user can create any new design or open an existing design. In the Cadence environment the Xilinx Design flow is presented as an ASIC design kit. In the schematic window, menus *Tools* ----> *ASIC Kit*, opens the ASIC Kit. ***Before opening the ASIC design kit, a design check and save operation must be performed.*** The design kit provides a graphical interface for processing Xilinx designs created with Composer. This graphical interface provides an on-screen flow chart that represents the Xilinx design flow. The ASIC kit for XC4000 designs is shown in Figure A.2. Each of the steps in the design flow is represented by a button. Upon a click, the button executes a script that runs programs necessary to complete a particular portion of the design process. Since this environment is script based, once the user becomes familiar

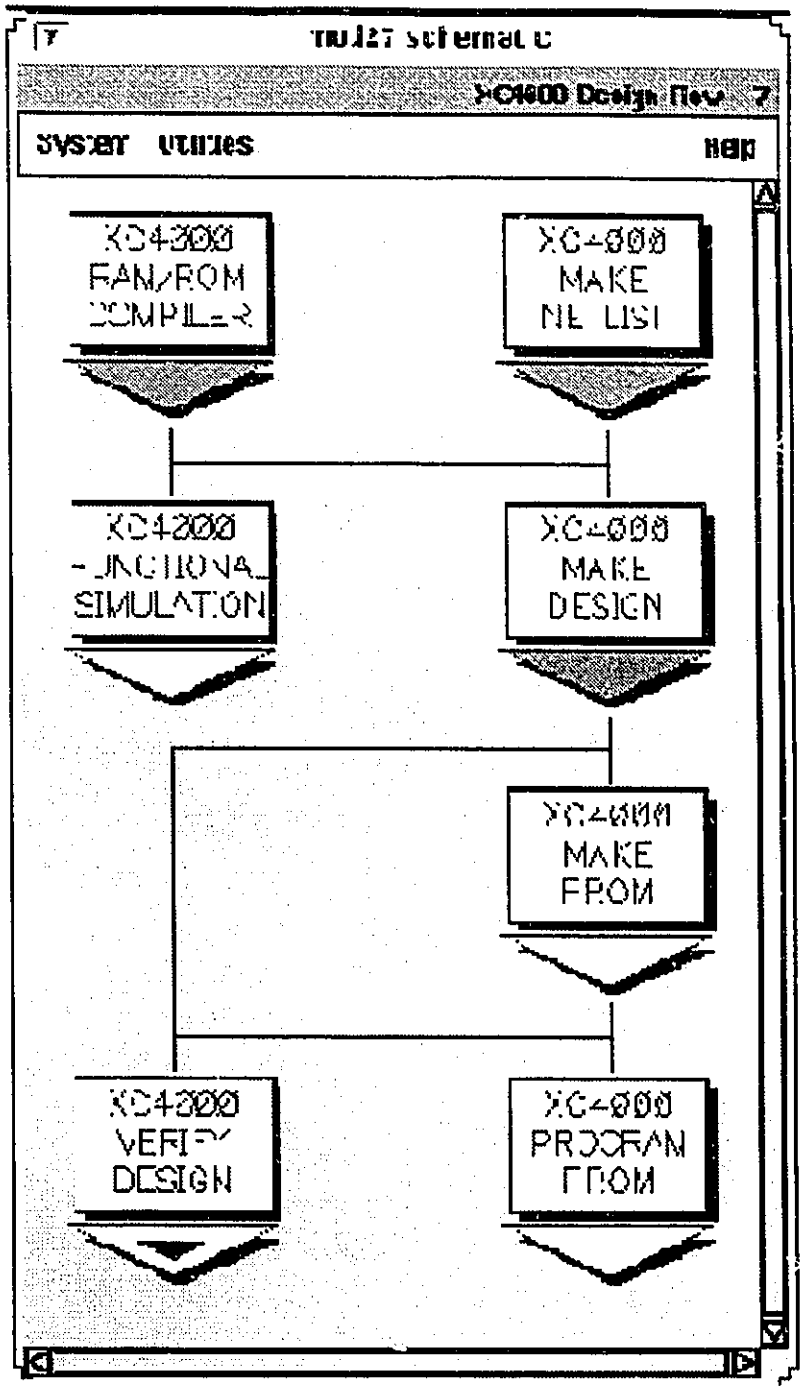


Figure A.2 ASIC Kit for XC4000 Implementation

with the Xilinx design process, she/he can alter the scripts to customize design processing to better fit her/his needs.

The designer, after opening the ASIC kit, *chooses for implementation one family of Xilinx FPGA* out of the three presently available. Depending on this choice, the corresponding flow chart is opened. It is important to note that due to this, *the designer has to decide before drawing the schematic, which family the design is to be implemented in, and must choose library components only from that family.* Updating from the chosen family to another is possible, but the design should have all the components only from that chosen family. Placing components in a design from more than one family library can pose problems. The design flow chart for each family is different, but the basic steps followed in every one of them are the same. After selecting the Xilinx family the user specifies the working directory, i.e the directory in which, the ASIC kit places the files[1] generated. In order to make use of the advanced features in the XC4000, some additional steps are present in the XC4000 flow chart. Steps involved in XC4000 flow chart are explained in detail below.

A.1.1 Make Netlist

In this step, *the part type and the speed grade of the target FPGA* within the particular family is selected. Then the `Make Netlist` button executes a shell script in an xterm window that runs the EDIFOUT program on the Composer design; then it runs EDIF2XNF on the resulting EDIF file. The EDIF models for the library components are stored in the $XACT/edif4000 directory. Make Netlist converts the schematic design information into a file format specified by Xilinx - Xilinx Netlist Format, the file generated has the suffix *xnf*.

Apart from the required *.xnf* file, the Make Netlist process creates a set of files (approximately 10), some of them are - the *.log* files for each EDIFOUT and EDIF2XNF

1. After one complete Xilinx design cycle, the user may have 35-45 files, so good file keeping is essential.

program, the *.sh* files for the shell scripts executed, the *.edn* file which contains the edif netlist. The number of files generated depends upon the schematic design, if the schematic design is hierarchical, then individual *.xnf* files are generated for every lower level schematic design, placed as a symbol in the top most level of the schematic. *Beyond this step, the rest of the flow chart (except for the simulation part) executes the software progams provided by Xilinx, in the XACT Development System.*

BUG or Feature? - Before generating XNF files, the user has a choice to specify, whether an unflattened or flattened netlist file has to be generated. Even if the user specifies the unflattened option, the shell script executes the program XNFMerge, which flattens the hierarchy, and the resulting *.xnf* file is a flattened file with no hierarchical information. However the Make Netlist process also generates a *.xnf* file with 'e' extension to the design name - *designe.xnf*, this file is an unflattened netlist file and this can be used when an unflattened netlist is required.

There is a problem generating XNFnetlist for schematic designs which are hierarchical. (i.e the schematic has symbols representing a lower level schematic). In the lower level schematics the input and output pins remain unconnected, as they will only be connected with signals in the higher level schematic. But MakeNetlist removes nets which are unconnected and produces a flattened netlist file with missing nets. *To resolve this problem, first create an unflattened netlist file, and then edit the xnf files of the lower level schematic to add the missing nets and invoke XNFMerge to flatten the netlist.* If the flattened netlist with missing nets is used, an error will be produced while executing PPR.

A.1.2 Make Design

Make Design push button calls for the Xilinx XMake Program. XMake automatically activates the translation programs needed to convert a *.xnf* file into an *.lca* file. There are several options in running the XMake program. First, the output files generated by the program and the options necessary are discussed.

design.xnf - XMake can completely flatten the top level XNF file of the design.

file.xnx - XNX files are unflattened XNF files, created by using the *-x option*. XMake creates XNF files for each module in the design hierarchy, and preserves them as files with suffix *.xnx*.

design.out - As XMake invokes various translation programs, it redirects their output to the file *design.out* . This is an ASCII text file containing all the screen output from the programs invoked by XMake. Since the .out file contains all warning and/or error messages that occur during the design processing, *the user should always review the .out file after XMake has run to ensure that the design is error -free.*

design.lca - XMake creates an LCA file that is partitioned, placed, and routed by the PPR (Partition, Place, Route) software.

design.bit - If PPR successfully routes the design, XMake creates a bit stream file by executing the MakeBits software, which can be downloaded into a FPGA.

design.mak - When XMake is run, it creates a text MAK file, that lists each software program executed including the options that were used by the translation programs. This is similar to a Makefile in Unix.

.log files for every module executed are always created. Some options for the XMake program are listed here:

-g - With this option XMake generates *.mak* file without performing any design processing. The option is generally used when the user wants to create a custom MAK file, but wants XMake to generate an initial script that can be edited.

-m - The MakeBits program which is executed after PPR to create bit files is disabled with this option.

-n - This option specifies XMake not to run PPR.

By default the XMake executes these programs: XNFMerge - to flatten the hierarchy, PPR - to partition,place and route the design, MakeBits - to generate a bit stream file. If something else other than the default is required, the user can specify options for the XMake, or the *.mak* file can be edited and that file can be executed. The *.mak* file lists the software executed along with the input and ouput file associated with the software.

A.1.3 RAM/ROM Compiler

This software aids the use of the on-chip RAM available in the Xilinx 4000 series FPGAs. For memory compilation the user has to specify: the memory cell name, the depth and width of the memory and the memory type (either RAM or ROM). The autogenerate symbol option is used to generate a symbol for the memory cell, a symbol is created in the memory cell name and it is placed in the specified target cell library. When the *OK* button is clicked, the Xilinx programs **mgaddr** and **memgen** are executed. After completion, *.mem* and *.xnf* files are created for the defined memory. The ROM symbol is used in the schematic. The properties of the symbol should be edited in the schematic editor. Add the property, Name=FILE, Type=String, and value=memory cell name.*mem* file. This attaches an FILE attribute to the symbol, the operation of the symbol is specified by the .mem file now. While creating a XNF netlist the program XNFmerge can be used to merge the *.xnf* file of the ROM with that of the rest of the schematic.

If the memory type is ROM, then there are some additional steps required because, for a ROM, the initial contents of the ROM have to be specified. Edit the *.mem* file to specify the contents of the ROM and do ROM compilation once again, this time the autogenerate symbol option is off. What is done here is that the ROM is recompiled with the initial values typed in the *.mem* file. Compilation passes the initial content information to the *.xnf* file. Placing the memory symbol and attaching the file attribute procedure is the same for both RAM or ROM.

A.1.4 Functional and Timing Simulation

These steps prepare the design for Verilog-XL^{TM} simulation. Timing simulation is a sub-menu in the verify design menu shown in fig. Both the simulation procedures are explained in detail in Appendix B.

A.1.5 Make PROM

The Xilinx FPGAs are built using static RAM cells, hence every time the chip is powered up, the configuration information has to be downloaded. To avoid this the XC4000 chip can be interfaced with a PROM, which can permanently store the configuration information. The Make PROM step executes the Xilinx Make PROM software, it converts the bit file into one of the three configuration formats: MCS-86 (Intel), EXORMAX (Motorola), or TEKHEX (Tektronix).

A.1.6 Program PROM

This button invokes the Xilinx XPP program which can be used with the Xilinx PROM Programmer to program a serial-configuration PROM with the files created in the MakePROM process. Currently, in the VLSI Lab, a Xilinx Programmer is not available.

A.1.7 XChecker Cable

If the user clicks the *verify design* menu with the middle button, another sub-menu appears with the above title. This software is used to download the bit file to an FPGA chip for configuration. Downloading is done through the Xchecker Cable, the cable is connected to one of the RS-232 communication ports (/dev/ttya or /dev/ttyb) of the SUN workstation. In this ASIC kit set up, the correct port is automatically selected.

A.2 LCA[1] Features in Cadence-Composer

There exist few differences between Xilinx LCA designs and other ASIC or board level designs, though LCA schematic design generally involves the same techniques as designs for other technologies. Most of these differences involve adding LCA-specific information to the schematic. This information is used by LCA implementation software.

These features are optional and they are useful for producing an efficient LCA design. The most commonly used LCA-specific properties are:

LOC and NOT_LOC - The location property or constraint specifies the locations within an LCA device in which flip-flops, I/O pads, etc. to be placed.

BLKNM - The block name property assigns a name to a flip-flop, I/O pad, CLB, or IOB. This is useful in understanding the *.lca* file when viewing it using Xilinx Design Editor.
Net Properties - These are properties assigned to nets, which affect the partitioning and placement of them.

First conventions for naming signals and symbols in LCA design are listed here:

1. Only names consisting of A-Z, a-z, 0-9, and "_" are permitted in user-defined names, other characters are not allowed, since this is the character set permitted by Verilog identifiers.
2. XACT reserved names should not be used as symbol names. For example pad names such as P1 and P2 should not be used.

A.2.1 Creating Properties

Follow the steps below to add properties to a symbol in Composer.

1. Select the symbol to which the property has to be added.
2. Select *Properties* from the Edit menu, the Instance properties pop-up form appears.

1. LCA - Logic Configurable Array.

3. Click on the Add button in the pop-up form. The Add property pop-up form appears.
4. Enter "sym_prop","pad_prop",or "net_prop" (depending on the property) in the Name field.
5. Select "String" in the Type field.
6. Enter the property and its value, *property_name=value*. Multiple properties are separated by commas.
7. Click on the OK button in the Add property pop-up form.
8. Click on the OK button in the Instance properties pop-up form.

In the following section some of the LCA-specific properties are explained in detail.

- **LOC and LOC_NOT Constraints**

These properties are used to specify locations for the associated logic.

LOC=blocks - Place the associated logic within the area defined by the *blocks*.

LOC_NOT=blocks - Do not place the associated logic within the area defined by *blocks*.

Soft macros and flip-flops can be assigned to a single CLB location, a list of CLB locations, or a rectangular block of CLB locations. The exact function generator or flip-flop within a CLB can also be specified. In the XC4000 series, CLB locations are identified as CLB_R#C#. For example the upper-left CLB is CLB_R1C1.

To attach a LOC constraint to a logic element, the property *sym_prop* should be used while following the procedure for creating properties. If the target symbol represents a soft macro the location constraint is applied to all flip-flops contained in the macro. If the indicated logic doesn't fit into the specified block(s), the constraint is ignored. Some examples of the constraints are given below:

LOC=CLB_R1C1 - Place logic in CLB_R1C1.

LOC_NOT=CLB_R1C1 - Do not place logic in CLB_R1C1.

LOC=CLB_R*C1 - Place logic within the first column of CLBs. The asterisk (*) is a wild card character.

LOC=CLB_R1C1;LOC=CLB_R1C2 - Place logic in CLB_R1C1 or in CLB_R1C2, there is no significance to the order of the LOC statements.

LOC=CLB_R1C1;CLB_R8C5 - Place logic within the rectangular block specified by CLB_R1C1 and CLB_R8C5.

LOC=CLB_R2C3.FFX - Place logic in the X flip-flop of CLB_R2C3, use FFY for Y flip-flop.

Location constraints can be applied to the I/O pads in the LCA design. I/O pads represent the physical package pins of the FPGA chip. There are I/O pads for inputs,ouputs and for bidirectional signals. An I/O pad has to be connected to the internal signals through an input or output buffer or through INFFs, OUTFFs , if the INFF or OUTFF elements are used then the I/O pad signal is connected through the flip-flops present in the IO blocks of the FPGA chip.Placing LOC constraints for I/O pads is similar to the procedure followed for CLBs. Some examples are given below:

LOC=P13 - Place I/O element in location P13.

LOC=T - Place I/O element in IOBs on top edge of the die. For the other three die edges ,use B(bottom), L(left), R(right).

LOC=LT - Place the I/O element in IOBs along the top half of the left edge of the die.

LOC<>T - Do not place I/O element on the left edge of the die.

In general, it is not advisable to place location constraints for I/O pads, as it may pose routability problems in the PPR software stage. However location constraints are advised if the user wishes to lock the signals in particular IOBs to aid in testing of the configured FPGA.

It can be seen that in order to specify location constraints, the user has to have a good understanding of the architecture details of the FPGA chip. It is advised that the user refer to the XC4000 Data Book and view the .lca file using XACT Design Editor.

- **BLKNM Property**

The BLKNM property is used to assign names to CLB and IOB primitives, and to basic logic elements such as gates and flip-flops. This can be very useful when viewing .lca file using Xilinx Design Editor.

To assign names to CLBs, IOBs,flip-flops or logic gates, edit the property of the element using the procedure explained before, and attach a *sym_prop* property with BLKNM=*name* to the element. If multiple gates or flip-flops carry the same names, then the PPR program attempts to partition these logic elements into the same block.

- **Net Properties**

Net properties are used to identify timing requirements of a net. Every net carries routing priority or "net weight" from 0 to 100. If no specific net weight is assigned, the default net weight of 1 is assumed.

To define a net property, attach a *net_prop* symbol (from the XilinxBasic library) to the desired net. Add the desired property using the *net_prop* option. Use commas, to separate multiple net properties.

Some of the choices are:

C - Critical, The C property identifies the net to be critical, a net weight of 100 is assumed.This is given the highest routing priority.

N - Non critical, This property identifies a net as non critical. A net weight of 0 is assumed and the lowest routing priority is given.

W - Weight (Relative Routing Priority), this property is used to assign a relative routing priority to the net. Values from 1-99 can be assigned.

X - External, The X property is used to identify a net as external. An external net is one that exists at a CLB output, and is not absorbed into a CLB. For example, if a net between a logic gate and a flip-flop is tagged external, then PPR places the combinational logic and the flip-flop in different CLBs, so that the external net exists at the CLB output.

A.3 LCA-specific schematic symbols

Global Buffers - There are special buffers in the Xilinx chip for critical signals such as clocks. They are called primary and secondary global buffers. If library elements *bufgp* and *bufgs* are used the primary and secondary buffers can be accessed. The primary clock buffer is faster than the secondary, but the secondary clock buffers can be driven by internal logic, whereas the primary cannot. The primary buffers can only be driven by the clock pins of the CLB. *So for better routability of the design, for signals like clock enable for flip-flops, the user may use secondary global buffers and the usage of primary global buffers can be restricted to clock signals alone.*

Oscillator - Xilinx 4000 series FPGAs have an internal oscillator, which can generate 5 different frequencies, 8 MHz, 500 KHz, 16 KHz, 490 Hz and 15 Hz. Although there are five outputs from the macro, only three can be used at any time, 8 MHz and two of the remaining four. An error occurs if more than three outputs are used simultaneously. The internal oscillator is accessed by using the library element OSC4. *The OSC4 outputs must be connected through buffers to the internal circuit.*

STARTUP Symbol - The STARTUP macro (Figure A.3) is used for global set /reset, global 3-state control, and user configuration clock. In XC4000 series FPGAs there is a dedicated reset net which can set or clear each register. This reset net does not compete with other routing resources. This net can be connected to any package pin as a global reset input. In order to access this net a input pad should be connected to the GSR pin of

the startup symbol. A location constraint can be attached to the pad, for the user to fix the reset input of the

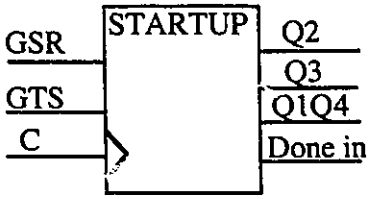


Figure A.3 STARTUP Symbol

FPGA chip. *After Configuration if* the *GSR pin is high, it sets or resets every flip-flop in the FPGA device depending on its initialization state. GSR does not clear the LCA's configuration memory.*

Following configuration, the global 3-state control (GTS), when High, forces all the IOB outputs into high-impedance mode. This isolates the device outputs from the circuit, but inputs are still active.

In the XC4000 FPGA chips, there is an I/O package pin for CCLK (Configuration Clock). This pin can serve as input or output depending on the way the FPGA is configured. If the FPGA is configured by downloading information using XChecker cable (serial slave mode) then the CCLK pin receives the configuration clock as input from the XChecker cable assembly. If the Xilinx chip is interfaced with a PROM (master parallel mode) then the CCLK pin outputs, clock for the FPGA configuration. The internal oscillator in the chip is used as the clock generator. When using PROM interface method, if the CCLK generated from the chip is not used and a user clock is provided for that purpose, then that user clock has to be connected to the 'c' pin of the startup symbol. Now the startup of the device is synchronized with the user clock. The startup outputs Q2,Q3,Q1,Q4 and DONEIN) display the progress/status of the startup process following the configuration. For more information on the output status, the user.should refer to XC4000 Data Book, StartUp section, where a detailed explanation with timing diagrams is provided.

A.4 The XACT Development System

The XACT Development System is supplied by the Xilinx company. In the Xilinx design flow, Cadence tools are used only for schematic entry and for obtaining the schematic information in XNF format, the rest part of the design flow is implemented by the software provided in the XACT Development System. The ASIC kit provided by Cadence executes these programs, all these programs can also be executed as command line instructions and they can also be used in the windows environment using Xilinx Design Manager (XDM).

XACT Development System consists of these programs for XC4000 design implementation:

XMake program

MemGen

XNFCvt

XNFUpd

XNFMerge

PPR

MakeBits

MakePROM

XDelay

LCA2XNF

BAX - Back Annotation

XACT Design Editor

The following peripherals and hardware are also included:

- XC4000 Design Demonstration Board
- XChecker Universal Download/Readback Cable

To invoke the Xilinx Design Manager[1], the user has to type

```
XDM &
```

This opens XDM's X-Window with menus, allowing the user to invoke any of the software packages. XDM maintains a *proglist.xdm* text file to minimize the time required to start up. The text file contains a list of XDM supported programs (executable files) installed on the system. When reading *proglist.xdm*, XDM searches for it in the following order, in the current working directory, in the $XACT directory and then in the data subdirectory within the $XACT directory. If the *proglist.xdm* file is not found in the above directories, XDM creates one and writes it in this order, $XACT directory, data subdirectory in the $XACT directory, in the current working directory.

A detailed menu tour of all the menus in the window is given in "XACT Development System, Reference Guide." However the user should note that there is one Design Entry Menu, in the XDM, that when the user clicks on it, generates a message: *no programs installed* - This is because the design entry is done separately using Cadence Composer and no Xilinx packages are used for design entry purpose. The user can use XDM only after obtaining the schematic information in XNF format using the ASIC kit in Cadence.

Every Xilinx program is described in the following section, some programs are given a more detailed explanation.

XMake, Memgen and MakePROM software were explained in detail in Xilinx Design Flow section of this appendix ("Make Design" on page 79,"RAM/ROM Compiler" on page 81, "Make PROM" on page 82 in SectionA.1.).

1. The user has to select the Xilinx-OPUS option when she/he logs into the SUN station, in this option appropriate path and environment variables ($XACT) is set and X-Windows is kept running which is required for XDM to operate.

A.4.1 XNFCvt

Xilinx has made modifications in the XNF syntax to include new design features, so, as of now, there can be three versions for XNF files. The version of the XNF file is given by the first line of the XNF file.

LCANET, 1 (this is a version 1 XNF file)
LCANET, 2 (this is a version 2 XNF file)
LCANET, 4 (this is a version 4 XNF file)

XNFCvt converts a version 4 XNF file to a version 1 XNF file, a version 4 to a version 2, or a version 2 to a version 1. *Version 4 XNF files that represent XC4000 devices cannot be converted to version 1 or version 2 XNF files.*

A.4.2 XNFUpd

XNFUpd generates a version 4 XNF file from a version 1 or version 2 XNF file. In the process of updating to version 4 XNF, a design can be converted from an XC2000 or XC3000 design to an XC4000 design.

A.4.3 XNFMerge

XNFMerge takes a hierarchical design and creates a "flattened" design, which contains no references to other XNF files. In the XC4000 design flow, XNFMerge combines XNF files from different sources. In most cases XNFMerge is not needed for XC4000 designs, since the PPR program will automatically read in any lower-level XNF files required. When XNFMerge is executed it creates a *.mrg* file, the *.mrg* file contains the following information, the files read in, the signals bound together, the number of signals, primitive symbols, and unresolved symbols in the output design. XNFMerge binds signals at different levels of hierarchy, by signal names and pin names.

A.4.4 PPR

Partition, Place and Route (PPR) program partitions, places and routes an XC4000 design. The input file is a .xnf file and the output generated is *.lca* file. PPR can be excuted from

the XDM or from an operating system prompt. Executing PPR from the operating system provides more control over PPR function than executing PPR from XDM.

Use the following syntax to execute PPR in the operating system prompt.

```
ppr design paramfile=textfile [parameter=value..]
```

A parameter file is a text file containing a list of desired parameters (options) and their respective values as in the following example.

```
estimate=true
justflatten=false
outfile=test
```

The additional parameters may be specified at the command line, these override similar parameters specified in the parameter file.

Input files for PPR:

design.xnf - PPR accepts an XNF file translated from an XC4000 design as input.

design.cst - PPR automatically reads this optional constraints file, if it exists. This file must have the same name as the input XNF file, but witha "cst" extension. This file can have location constraints for logic elements.

Output files generated by PPR:

design.lca - PPR generates a placed and routed *lca* file. This file is given the input design name (with an lca extension).

design.lcb - An existing *lca* file is changed to an *lcb* file at the start of the each routing phase; a new lca file is subsequently created.

design.bid - The Block Implementation Detail file, created during the routing phase, contains information on how the design is partitioned. ***It contains net name information needed by LCA2XNF and BAX for the back-annotation process.***

design.bib - An existing *bid* file is changed to a *bib* file at the start of each routing phase; a new *bid* file is subsequently created.

ppr.log - PPR produces a file named ppr.log that contains all information output to the screen during PPR execution.

design.rpf - PPR generates an rpf file on its first routing pass. This file includes design statistics such as pin locations and logic reduction information.

design.rpt - PPR produces a report file that includes design statistics, pin locations, and information on logic reduction.

In the above listed files, the most important files are the LCA, BID, RPT files. *The user must go through the rpt file after completion of PPR, if there are any unrouted signals, the rpt file provides a report on them.*

There are so many (around 40) options associated with PPR program. With the default settings of the options, the user can manage to get a partitioned, placed and routed design. There are options for placement improvement, routing improvement and to set up timing constraints for signals between two flip-flops, or between two I/O pads. PPR options can be specified in the *xactinit.dat* file, every time PPR is executed it checks for options in that file. An *xactinit.dat* file is automatically installed when PPR is installed in the $XACT/data directory. The parameters defined in this file apply to all users running on the entire system. The user can have an edited xactinit.dat file in the current working directory to create a local profile that applies to her/his designs. This file overrides the xactinit.dat in the $XACT/data directory.

Past practice with PPR shows that it often routes only 93%-97% of the design and the designer has to route the remaining portion using XACT Design Editor.

Routing Tip: The designer should use secondary global buffers more than the primary global buffers. The use of primary global buffers should be restricted to very critical signals such as clock signals. Signals such as clock enable can be routed using secondary global buffers. In general, routing resources for secondary global buffers are more flexible.

A.4.5 MakeBits

MakeBits is used to create configuration bit streams for LCA designs. The configuration bitstream describes the internal logic functions and interconnections of an LCA device.

There are two functionally equivalent versions of MakeBits. One is invoked as part of the XACT Design Editor (XDE), and the other is a stand-alone version, that can be run from the operating-system shell. The two versions differ only in their user-interface; both produce an identical BIT file.

After the design has been placed and routed using PPR, the LCA device has to be configured by filling its configuration memory cells with appropriate binary information. MakeBits converts an LCA design file, i.e., a fully routed design, and generates a binary bitstream file that contains the configuration information for an LCA device. MakeBits can also be used to generate other types of files and reports, containing information about timing and design rule violations.

The PPR program that generates the original routed .lca design file, does not write delay information into the file. *MakeBits when executed with -w option, writes out an .lca design file with timing information over the input file. This is necessary for performing timing simulation.*

If MakeBits is executed with -d option, it runs the XACT Design Rule Checker, this is useful particularly when some manual routing is done in the design. A design should be free of any DRC errors for MakeBits to generate a bit stream. If -d option is used no bit stream is generated, after DRC is run and no errors found, re-invoke MakeBits without the -d option.

Input file for MakeBits:

Input file to the MakeBits progarm is the fully routed *.lca* file.

Output files generated by MakeBits:

design.bit - This binary (0s and 1s) bit file contains the configuration data for an LCA design.

design.lca - If the -w option is used, MakeBits writes an lca design file with the same name as the input design, with the addition of timing information. Adding delay information allows timing simulation to be performed.

design.rbt - If the -b option is used, MakeBits creates an ASCII version of the configuration bit stream.

A.4.6 XDelay

The XDelay program is used to obtain a detailed timing information about a particular *.lca* design. It is not a simulator, or a design rules checker that looks for logic errors in the design. XDelay can be used to create a detailed list of delays for all paths, so that an estimate of the system performance can be obtained.

XDelay can be called from the command line or the user may use a graphic interface to invoke commands. To use the XDelay graphic interface, type "xdelay".

XDelay can be asked to give a delay report for different kinds of paths in the design,

-ClockToPad: If XDelay is invoked with this option, it will report paths that start at clocked outputs, like flip-flop Q outputs, and end at output pads.

-PadToPad: Reports only on combinatorial paths with no clocked elements

-PadToSetUp: Reports only paths that start at input pads and end at clocked elements (flip-flop data input)

-ClockToSetUp: Reports only paths starting at clocked outputs (such as flip-flop outputs) and ending at clocked inputs, such as flip-flop data inputs.

This is a very useful option in the case of systolic pipelined designs, as the longest path between any two flip-flops determines the clock rate of the system. So XDelay can be performed with this option to get the longest path delay between any two flip-flops and the estimated operating speed of the system can be calculated.

An example of Xdelay usage is given below

```
xdelay design.lca -o report -t time.xtm
```

This tells XDelay to perform timing analysis on the specified *.lca* file, and to redirect its output to a file named *report*, and it should use the options specified in the template file *time* with suffix *.xtm*. The template file would typically be as shown:

Xdelay -ClockToSetup
Xdelay -FromAll
Xdelay -ToAll

These commands tell the XDelay software to give a report on paths between any two flip-flops and they also force XDelay to consider all the flip-flops. The result of this can be seen in the file named *report*, it is observed that XDelay did not list all the paths between two flip-flops but just listed the longest path between two flip-flops. Typically a design may have thousands of paths, and it may take hours to process an entire design. The contents of the resulting *report* file is given as a sample:

```
Xdelay timing analysis options:
From All.
To All.
Report file may include Clock To Set up Paths.
Clock net "CLK" path delays:
 Clock To Setup         : 34.2 ns (2 block levels)
Clock to Q, net "MFOY<0>" to FF Setup (D) at FIR2.G2
     Target FFY drives output net "FIR1"

      Minimum Clock Period : 34.2 ns
      Estimated Maximum Clock Speed: 29.3 MHz.
```

The file gives the longest path and identifies that net in the design.

A.4.7 LCA2XNF

The LCA2XNF program converts an LCA file to an XNF file that can be used for timing simulation. The resulting XNF file contains all worst-case block and net delays which can be used to generate necessary simulation files for Verilog-XL^{TM} program. The required input files are the *.lca* and *.bid* files. The Block Implementation Detail (bid) file is generated by the PPR program, the *.bid* file lists the original names of nets at the outputs of

function generators and at the pads. LCA2XNF uses this information to create signal names.

A.4.8 BAX - Back Annotation Program

The BAX program can improve the simulation interface by creating an XNF file that contains delay and original schematic information. With BAX, the user can use most of the original signal names for timing simulation.

BAX needs the *.xnf* file generated by the LCA2XNF program as input and it also needs the *.bid* file generated by PPR. BAX restores original names for output signals of function generators and pad signals. *BAX does not restore the original names for signals inside the function generators.*

A.4.9 XACT Design Editor (XDE)

This program can be invoked by typing "xact" or "xde" at the operating system prompt or at the XDM command line. Upon typing, an executive display window appears, it contains commands to load and save designs and to invoke any XACT subprogram (MakeBits, MakePROM, EditLCA). The XDE program checks the $XACT/data/xact.pro file when opening the graphics window. It reads default settings for its environment from the *.pro* profile file. If a user has a xact.pro file in the current directory, that overrides the one in the $XACT/data directory.

This document focuses on the EditLCA feature of the XDE. The EditLCA design editor is a graphics program that displays and manipulates an electronic image of the functional layout of an LCA (Logic Cell Array). It consists of the following :

Physical Interconnect Editor (PIE)

Block Editor

Physical Interconnect Editor (PIE) Display

The PIE consists of the following:

World View (an optional map of the entire LCA)

LCA Layout

Cursor

Cursor status line

Pull-down menus

Message line

Command line

LCA layout shows the CLBs and IOBs in the FPGA chip. Each CLB is identified by the row and column numbers in XC4000 series. If a routed *.lca* file is viewed, then the connections between the CLBs and with IOBs are shown. World View is a rectangular box which overlays on the LCA layout, it shows all the blocks in the LCA device. Display of the world view can be turned off or on using the *Show* command. The global clock buffers and the oscillator are displayed on the LCA layout diagram.

If a user wishes to do manual routing she/he has refer to the XDE section of the XACT Development System, Reference Guide for the following commands: Addnet, Addpin, Editnet, Delnet, Namenet, Route, Autoroute.

Every CLB or IOB can be edited using the Block Editor Display, the command EditBlk should be used.

The user in the command line of the block editor display can specify equations to configure each function generator of the CLB.

There are several features associated with XDE, and again the user can refer to XACT Development System, Reference Guide for detailed explanation. The features discussed here will be the most used.

A.4.10 XChecker Universal Download Cable

The XChecker™ cable and software are designed to work with all the families of the Xilinx FPGAs. They are used to download, readback, and verify configuration data, as well as to probe internal logic states of the XC2000, XC3000 and XC4000 designs.

Xchecker cable is connected between the SUN station and the target FPGA in RS-232 Communication mode. The XChecker cable and software support the following capabilities.

It allows downloading of a design to the LCA on the target system.

After configuring an FPGA, XChecker can verify its configuration by comparing it to the original design.

The user can probe an LCA's internal logic with XChecker to debug designs. Probing is the execution of a readback of all the configuration data and extracting the internal logic states of desired signals from it.

XChecker hardware includes a DB-9/DB-25 adapter, the cable assembly with internal logic, a supplied test fixture, and a set of headers to connect the cable to the target FPGA device.

The Xchecker cable assembly houses internal circuitry consisting of a Xilinx FPGA, a static RAM, and an oscillator circuit. The internal Xilinx FPGA functions as an interface, between the Xchecker software and the target FPGA. The static RAM stores the configuration data for download and readback. The oscillator circuit provides a system clock and allows download and readback of configuration data.

The Xchecker cable transmits configuration data to all target LCAs at 921 kHz. In XC4000 devices, readback can be performed at three different clock rates: 921 kHz, ~2.75 MHz, and ~5.5 MHz. Communication between the Xchecker cable and the workstation would be typically at a 38.4 Kbaud

XChecker Software, can be invoked from the system shell or from XDM. The appropriate settings for the communication port and the baud rate can be done in the *xchecker.pro* file.

A.5 Documents for Reference

For a user to design schematics for FPGA implementation, it is advised that the user reads the *"The First Canadian Workshop on FPGAs, Xilinx Hands-On Workshop"* manual, developed by Dept. of Electrical and Computer Engg., University of Manitoba. It gives a menu by menu instruction for a complete design cycle and it also expalins the procedure for including RAM/ROMs in the schematic design.

Openbook On-line documentation in the workstation is another manual for reference. To open this type "openbook", the openbook appears with the Main Menu, then click the button for *Programmable IC Logic* Design, and then click *Xilinx Interface to Composer.* If information is needed for simulation click *Xilinx Interface to Verilog-XL.*

For detailed information on the Xilinx programs refer to *XACT Development System, Reference Guide.* The features of the Xilinx FPGAs are explained well in the *Xilinx Data Book.*

Appendix B

Xilinx Interface to Verilog-XL

Verilog-XL is a trademark of Cadence Design Systems., Inc. and it is the simulation tool used for simulating Xilinx designs. This appendix presents some notes on how to simulate a Xilinx design using Verilog-XL. There are two types of simulation, Functional and Timing, both of these methods are explained.

B.1 Functional Simulation

Functional simulation provides an effective method to verify the functionality of the design before the design has been placed and routed. This saves debugging time later in the design process. This can be referred to as unit-delay simulation, as unit delay times are used rather than actual delay times. Each logic gate is modelled to have 0.1ns unit propagation delay and all nets are considered to have 0ns routing delay.

The flow chart for functional simulation of XC4000 designs is shown in Figure B.1. To simulate a schematic design functionally, the design information must be translated into a Verilog^{TM} netlist file. First, an XNF file must be generated, it should be completely flattened and it should not be hierarchical. PPR when executed with *just flatten* option produces a completely flattened file without performing any placement or routing. The next step is to execute XNF2EDIF with *-v* option. The *-v* option is to create a legal Verilog netlist, Verilog-XL dictates that all net, instance and cell names be unique, only alpha numeric characters and underscores be used and names begin with a letter or underscore. If XNF2EDIF finds any illegal names it changes them. For example, the net name *out(4)*

would be changed to *out_4_*. After creating an EDIF netlist, execute, EDIF2VERILOG with *-v* option, this creates two

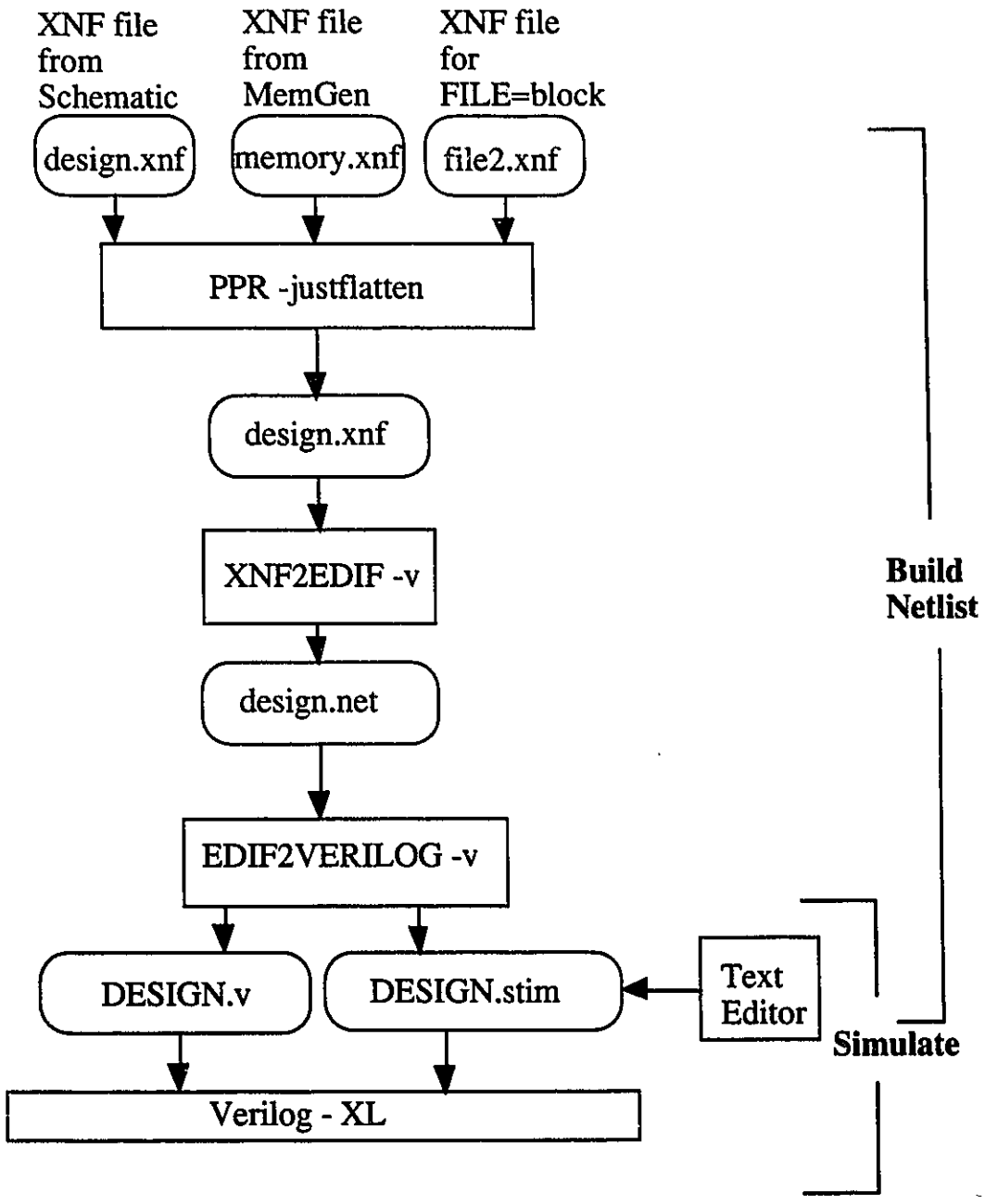


Figure B.1 Flow Chart for XC4000 Functional Simulation

files, *.v* and *.stim* file, *.v* is the Verilog netlist file and.*stim* is the stimulus file where the user can include test vectors. The *.stim* file is created with necessary module definitions and declarations, the user has to just add test vectors to the *.stim* file.

Verilog with *.stim* and *.v* files performs the simulation and gives the output as waveforms in a graphic window. The *gr_waves* waveform display scheme is used. Verilog also prints the output values shown by the waveform in the screen.

After obtaining the flattened XNF file, the above flow chart can be executed using the FUNCTIONAL SIMULATION push button in Composer ASIC kit or these commands can be executed at the command line by typing them. It is suggested that the user uses the ASIC kit to generate *.v* and *.stim* files initially and then invoke Verilog at the command line for subsequent simulations. The command is shown below:

```
verilog -f /cmc1/OPUS42/etc/license/verilog.vc
    -y /usr/local/engn/cmc7/422/tools/xilinx/data/verilog4000
    +libext+.v +notimingcheck DECODERF.v func.stim
```

In the above command, the *-f* option specifies the Verilog password file, *-y* option specifies the directory where the verilog models reside, *+libext+.v* specifies the file extension of the models. *notimingcheck* option indicates that the simulation is only functional and no timing check has to be done. Functional simulation is performed on the design named DECODER[1] and the test vectors are provided in the *func.stim* file.

B.2 Timing Simulation

Timing simulation verifies the design functionality by using worst-case delay information from the routing and block delays in a placed and routed LCA file. Once a design is partitioned it is no longer defined in terms of logic gates, it is defined only in terms of CLBs and the interconnection between them. For timing simulation some extra steps have to be performed. The flow chart for timing simulation is shown in Figure B.2.

1. If the .v netlist file is generated using the Composer ASIC kit, it adds an F extension to the design name.

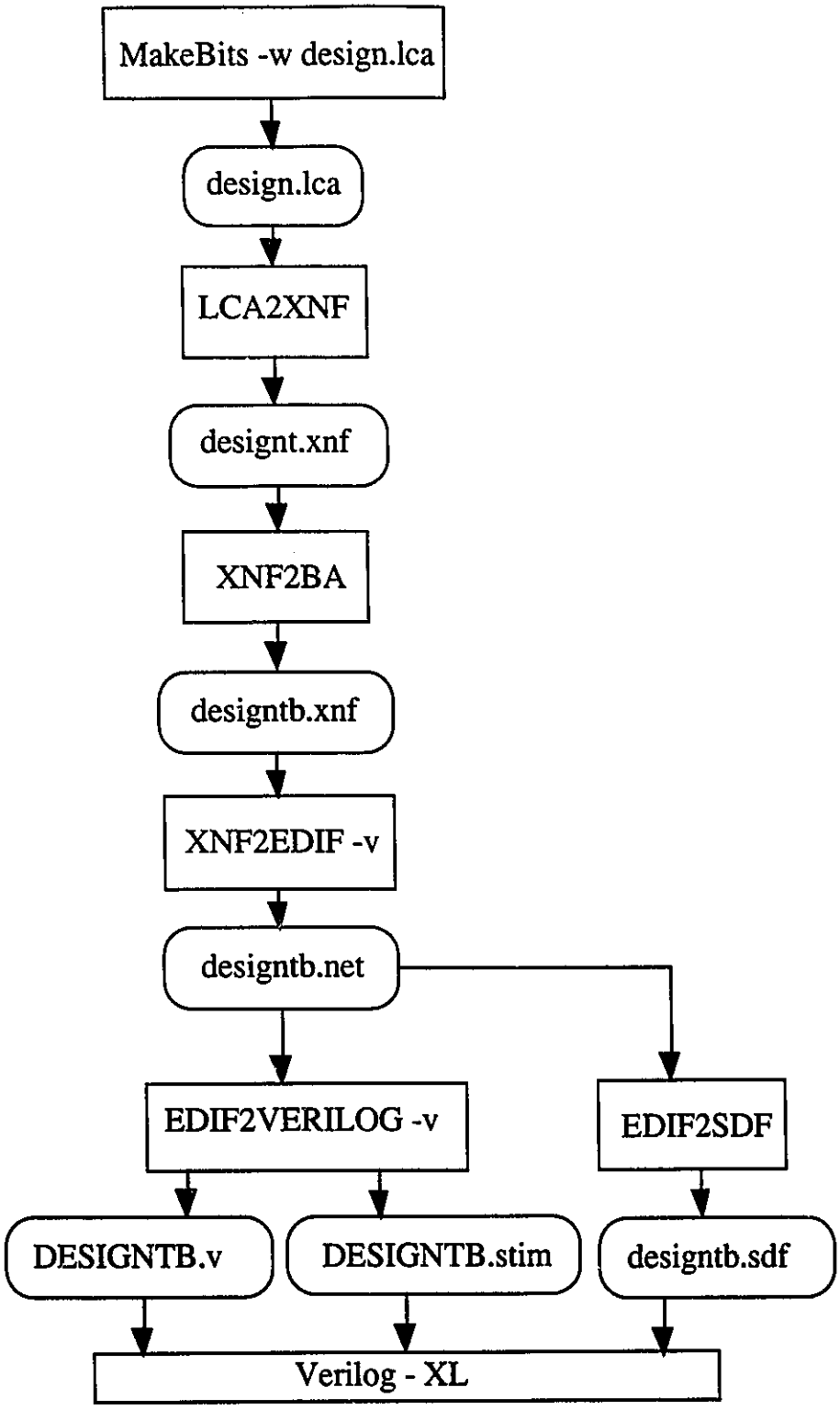


Figure B.2 Flow Chart for XC4000 Timing Simulation

The .lca file generated by PPR does not contain any delay information. So ***the user must execute MakeBits with -w option, this writes the delay information back into the LCA file.*** *Note that the flow chart given in the Verilog-XL manual (On-line documentation, Openbook) does not show this, and it is very important to perform this step, otherwise the whole timing simulation procedure is wrong. Timing simulation is mainly done to get an estimate of the operating speed of the system and if it is performed without considering any routing, there is no point in performing timing simulation. It is guessed, that the manual assumes that the user performs the MakeBits step.*

Now, LCA2XNF is executed. It translates the design, along with its delay information, into an XNF file. The next step is to perform XNF2BA (Back-Annotation), after a design is partitioned and placed, some of the original schematic information is lost in the process. XNF2BA combines an unrouted XNF file (given as input to the PPR) and a XNF file which contains the routing delay information (obtained after executing LCA2XNF, *designt.xnf*) and produces a new XNF file that has routing delays, original symbol names and original signal names. For simulation, if observing only the inputs and outputs of the design is of major interest, and if details of original schematic information are not needed, the user can skip the XNF2BA step. Moreover, net names for flip-flop outputs remain the same even after Partition, Place and Route. So for pipelined designs where only flip-flop outputs are of interest, the XNF2BA step can be skipped.

Then XNF2EDIF and EDIF2VERILOG with -v options are now performed. This produces the Verilog netlist *.v* file and the stimulus *.stim* file. The program EDIF2SDF extracts the timing information from the EDIF netlist and writes it into the Cadence Standard Delay Format (SDF) which is readable by Verilog-XL. The *.stim* file has the following command which reads in the SDF file.

```
$sdf_annotate ("designtb.sdf", test.t1);
```

Where *designtb.sdf* is the SDF file created by the EDIF2SDF program. *test.t1* refers to the scope (instance *t1*, module *test* would be defined in the *.v* and *.stim* file).

Timing simulation can be performed using the Composer ASIC Kit, or by invoking each one of the programs at the command line. As in functional simulation, it is suggested that the user use the ASIC kit to generate *.v*, *.stim and .sdf* files initially and then invoke Verilog at the command line for subsequent simulations. The command is shown below:

```
verilog -f /cmc1/OPUS42/etc/license/verilog.vc
     -y /usr/local/engn/cmc7/422/tools/xilinx/data/verilog4000
     +libext+.v DECODERT.v time.stim
```

Note that the *notimingcheck* option used in functional simulation is not included here. The required *.sdf* file is read by the *.stim* file and it is not specified with the command.

B.3 Verilog-XL - Miscellaneous

In the simulation process, an XRF file is created, XNF2EDIF modifies net and instance names to ensure that they are legal in Verilog. The XRF file contains information on renaming of net and instance names.

B.3.1 Global Set/Reset

In XC4000 FPGAs there is a global set/reset dedicated net, which can be accessed by using the STARTUP symbol. This can be used to set or reset any flip-flop in the design. If the symbol is placed in the schematic and an input pad is connected to the GSR pin, then this information is transferred to the Verilog netlist file. Even if the STARTUP symbol is not placed in the schematic, Verilog, in its netlist and stimulus file (for both functional and timing simulation), introduces a net named GLOBALSTRT for the global set/reset net. In the stimulus file, the GLOBALSTRT net should be pulsed to simulate the Power On Set/Reset as well as for re-initializing the flip-flops. The lines to be included (in the *.stim* file) are shown below:

```
initial
      begin
      GLOBALSTRT=0;
      #5;
      GLOBALSTRT=1;
      #5;
      GLOBALSTRT=0;
```

end

Here the GLOBALSTRT is pulsed, the duration of 5 ns[1] is approximately chosen.

B.3.2 Standard Delay Format

A part of a SDF file is presented in order to explain the format. The TIMESCALE for the SDF file is 1ns.

```
(DELAYFILE
        (DESIGN "DECODERT")
        (DATE "Thu Oct 6 23:29:43 1994")
        (VENDOR "Cadence Design Systems")
        (PROGRAM "edif2sdf")
        (VERSION "1.6")
        (DIVIDER)
        (VOLTAGE)
        (PROCESS)
        (TEMPERATURE)
        (TIMESCALE 1ns)
(CELL
     (CELLTYPE "FDRD_4K")
     (INSTANCE D1642)
     (DELAY (ABSOLUTE
          (DEVICE Q (5.00:5.00:5.00) (5.00:5.00:5.00))
          (PORT C (3.10:3.10:3.10) (3.10:3.10:3.10))
          (PORT RD (4.00:4.00:4.00) (4.00:4.00:4.00))
          (PORT CE (0.00:0.00:0.00) (0.00:0.00:0.00))
          (PORT D (0.00:0.00:0.00) (0.00:0.00:0.00))
          (PORT GSR (0.00:0.00:0.00) (0.00:0.00:0.00))))
     (TIMINGCHECK
          (SETUPHOLD D (posedge C) ((0.00)) ((6.00)))
          (SETUPHOLD RD (posedge C) ((6.00)) ((0.00)))
          (SETUPHOLD CE (posedge C) ((7.00)) ((0.00)))
          (WIDTH (posedge C) (5.00))
          (WIDTH (posedge RD) (5.00))))
```

The keyword CELL indicates beginning of an instance definition.DELAY denotes input and output port delay values. TIMINGCHECK is performed for any violation of setup and hold time of the flip-flop. WIDTH indicates the pulse width requirement value. In timing simulation, if the setup or hold time specification of any flip-flop is violated, Verilog outputs that to the screen as a warning message. It also specifies the INSTANCE name in the message. The user should watch for these warning messages when performing timing simulation.

1. The time scale for the stimulus file was 1 ns so 5 time units is equal to 5 ns.

B.3.3 How to probe internal signals?

The netlist file generated from EDIF2VERILOG contains the required *reg* declarations for design inputs and *wire* declarations for outputs of the schematic design. To probe intermediate signals, if the designer uses the *cmos4s* technology, probes can be added into the schematic; those signals are monitored and their outputs can be observed in the graphic window. That procedure does not seem to work for Xilinx technology, so an alternate technique is presented here. If the internal signals have to be probed, the designer has to add modifications to the *.stim* and *.v* files. In the schematic, the design inputs and outputs are connected to *ipad, opad* elements. The modifications made in the .v and .stim files are equivalent to attaching an *opad* (output pad) to the intermediate signal required to be probed. For example consider a design by name DECODER and let INTER be the signal to be probed. Follow these steps:

In the .v file, in the `module DECODERF`[1] declaration include INTER, this makes INTER an output from the design:

```
module DECODERF
(CLOCK, IN1,IN2, OUT1,OUT2, INTER);
```

Add the declaration`output INTER;` along with the other output declarations.

In the *.stim* file add the declaration wire INTER; and include the net INTER declarations in `DECODERF t1`, `$monitor`, and `$gr_waves` sections of the file.

If the user does not want to observe all the output signals, some output signal declarations can be omitted in the`$gr_waves` section.

It is advised that the designer give mnemonic names to every net in the schematic design, especially for flip-flop output nets and for input, output nets connected to ipads and opads, otherwise the netlist will contain unique numbers for nets and this makes understanding the netlist difficult.

1. Recall that EDIF2XNF adds an extension F to the design name if it is performing translation for functional simulation.

B.3.4 A bug in the comment line of the *.stim* file

In Verilog simulation for Xilinx designs the time unit used is 1 ns, this is specified by the timescale compiler directive, it gives the unit of measurement for time and delay values and the degree of accuracy for delays in all the modules.'`timescale 1 ns/100 ps` -This statement can be seen at the beginning of the *.stim* file. The *.stim* file created by EDIF2XNF initially has one set of input test vectors, with all inputs assigned "0". Consider IN1,IN2,IN3 as inputs in a design, the .stim file would have these lines,

```
initial
      begin
      IN1=0;
      IN2=0;
      IN3=0;
#10000; $STOP; //[1]Change data every 1MHz.
```

So 10,000 ns is equivalent to 10 ms and in the frequency scale it is 0.1 MHz. But EDIF2VERILOG places a comment that 10000 ns is equivalent to 1MHz. As this is just the comment line the user can ignore this statement. This is brought to the user's notice, in order that the user not be misled by this statement.

B.4 Documents for Reference

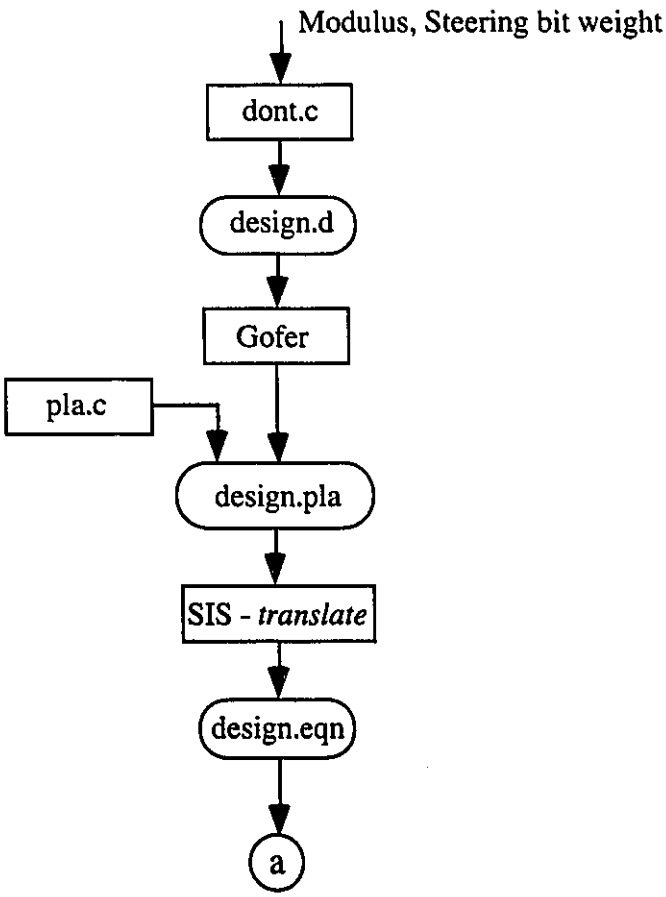
The documentation available for reference, is the *Openbook*, On-line documentation, in the SUN workstation. For the user to first understand Verilog-XL, the *Verilog-XL Tutorial*, present in the *Digital Logic Simulation* section is suggested. For the Xilinx interface manual, the user has to go to the *Programmable IC Logic Design* sub-menu from the *Main menu*.The *Xilinx Interface to Verilog-XL* manual can be accessed from there. *Verilog-XL Reference* manual in *Openbook* can also be considered.

1. Any line beginning with // is a comment line,

Appendix C

Software Flow for Minimized ROM Procedure

This section presents the flow chart (refer Figure C.1) of the software programs used for the Minimized ROM procedure. Every step in the flow chart is explained.



The flow chart is continued in the next page.

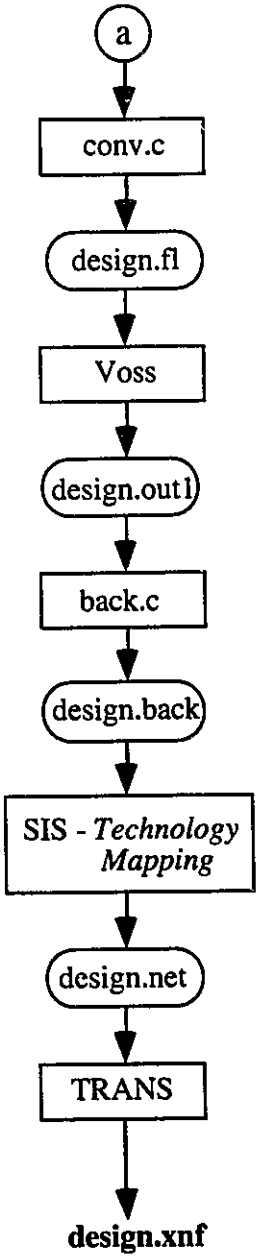


Figure C.1 Flow Chart for Minimized ROM Procedure

1. The inputs to the software flow are the required modulus and the steering bit weight, say modulus 27, steering bit weight 4, the program `dont.c` produces a truth table output readable by `Gofer` program. The resulting output file is given the extension*d*.

2. The Gofer program executes the don't cares elimination algorithm and prints the output values in the screen. The program `pla.c` is executed to create the truth table in the SIS, *pla* format, the *design.pla* file would have don't cares in its truth table, these don't cares are substituted with the output values given by the Gofer program. Now the *design.pla* file contains truth table information with don't cares values specified by the Gofer program.

3. The SIS package from University of California, Berkeley, supports several formats such as *pla*, *eqn*, *blif*. It also has conversion programs between those formats. So SIS is used for converting the *pla* format into *eqn* format. The resulting file is given the extension *eqn*.

4. The program `conv.c` converts the *eqn* format into a format required by Voss, a package used for BDD reduction. Voss was developed at the University of British Columbia. The file created has the extension *fl*.

5. Voss package gives its output only on the screen, it can not be redirected into a file, so its outputs are cut and pasted in a file which has an extension *outl*.

6. Voss output is not readable by SIS, so once again a conversion is performed. `back.c` converts the Voss output format into the *eqn* format given by SIS.

7. SIS is used for technology mapping. The following script is executed.

```
read_library ~/xilinx/SIS/radha.genlib
read_eqn design.back

xl_part_coll  -m -g 2 -c 50 -n 4
xl_coll_ck -n 4
xl_cover -e 60 -u 200 -n 4
xl_partition -m -n4

map
write_bdnet design.net
```

SIS requires a library in *genlib* format for technology mapping. The output of the SIS is given in *bdnet* format.

8. The output from SIS is converted to the required *xnf* (Xilinx Netlist Format) using TRANS a software created in University of Calgary.

Vita Auctoris

Radhaselvi Venkatesan was born on July 31, 1970 in Madras, India. She received her Bachelor's degree in Electronics and Communication in 1990, from Government College of Technology, Coimbatore, India. She worked for two years as a Research and Development Engineer at Pace Elcot Automation Limited, Madras. She started her Master's program in Electrical Engineering at the University of Windsor in Fall 1992. She will be joining Northern Telecom, Ottawa, in the Design for Testability division.